hesis
1964
#2

THE UNIVERSITY OF ALBERTA

# PROBABILISTIC AND DETERMINISTIC ASPECTS OF DIGITAL COMPUTERS

by

William S. Adams

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE.

DEPARTMENT OF MATHEMATICS
EDMONTON, ALBERTA
NOVEMBER, 1963.

## ACKNOWLEDGMENTS

My sincere thanks to Dr. John McNamee for, among other things, supervising this thesis, and to Dr. D.B. Scott, for providing a congenial and stimulating environment in which to work.

## ABSTRACT

This thesis surveys three main topics, information theory, coding theory and the structure of digital machines. These topics represent the main theoretical lines of inquiry into the concept of information.

The probabilistic assumptions and methods which form the basis of information theory are presented and developed as far as the fundamental theorem for finite discrete noisy channels. This theorem guarantees that information can be transmitted without error despite the presence of noise but produces impractical encoding methods.

Coding theory offers less powerful but more practical error-correcting codes based on deterministic aspects of information. Modern algebra provides methods of analysis for these codes and their mathematical development and implementation in terms of simple electronic devices are discussed.

The same electronic devices are used in the construction of digital computers which may be displayed as complex sequential sets of internal transfers of information. The static and dynamic structure of a modern digital computer are analysed by means of directed graphs and an algorithmic language.

The thesis attempts to unify several distinct lines of inquiry and to trace their significance in the analysis of existing digital computers and in the design of new computers. The analysis by directed graphs in Chapter 5 is believed to be new.

TABLE OF CONTENTS

Table of Contents (Con't)

Table of Contents (Con't) Page

# CHAPTER 1

## Introduction

This thesis surveys three of the main theoretical fields which have or will have a direct influence on the development of digital computers. It is addressed to those people who are aware of the uses of digital computers and who acknowledge the need for an understanding of the computers themselves at a level beyond the purely descriptive. The uses of digital computers are not considered except incidentally although these uses do have an influence on the development of computers.

The three fields are

1. Information Theory,
2. Error-Correcting Codes,

and 3. The Design of Digital Computers.

Each of these fields has a vast and growing literature despite their relative newness and reasonably complete bibliographies would be many times larger than this thesis. A fairly complete bibliography may be found in the following texts, "An Introduction to Information Theory" by F. M. Reza, "Foundations of Information Theory" by A. Feinstein, "Error-Correcting Codes" by W. W. Peterson and "Theory and Design of Digital Machines" by T. C. Bartee, I. L. Lebow and I.S. Reed.

Feinstein's book provides a rigorous treatment of the probabilistic aspects of the transmission of information. Bartee, Lebow and Reed present a remarkably lucid study of the deterministic aspects of the manipulation of information in terms of Boolean concepts.

Information theory, a rather independent offshoot of probability theory, has been put on a rigorous mathematical basis and is well advanced as a professional field in mathematics. Coding theory, nominally a part of information theory, is somewhat directed to implementation of isolated codes for the practical transmission of information but is in the process of changing over to more general mathematical procedures. The design (and theory) of computers is the least mathematically-oriented field and no fundamental organisation of the many aspects of the field seems imminent. Contributions to all three fields have come from an unusually wide spectrum of professionals, e.g., mathematicians, physicists, engineers and philosophers, to mention the more important contributors. While this wide viewpoint confers certain benefits, as Bartee (p. 270) says "one might wish, with Leibnitz, for a more universal language, which would allow for easier communication between members of the different groups studying digital machines, and a subsequent integration and broadening of the overall

theory."

As no overall mathematical integration was available, it seemed that a viewpoint was essential, if only to keep the thesis to a reasonable size. The attitude taken was that it was more important to present the concepts, assumptions and simplifications which could be used as a basis for mathematical development, than it was to explore the mathematical consequences of these concepts.

The thesis consists of six chapters. The first is introductory. The second deals with elementary codes which are in use or have been used in connection with digital computers. Chapter 3 presents an outline of information theory leading up to the fundamental theorem for discrete noisy channels. The binary symmetric channel is discussed in detail as it is important in the subsequent chapter on error-correcting codes and is by the far the most studied in the literature. Roughly the same material is covered by Feinstein in a more rigorous form in about 80 pages of his book. Chapter 4 is a description of the main results in encoding for the binary symmetric channel and their potential application to communication and digital computers. Chapter 5 is an attempt to describe the internal structure of a modern digital computer, the IBM 1620, in terms acceptable

to a mathematically-minded user. Chapter 6 discusses some topics of theoretical interest in all three fields.

The objective of the thesis is to present the digital computer against a theoretical background which, hopefully, will show that the study of the computer itself, as opposed to its applications, promises to be an unusually rich source of ideas (and problems) for many fields of mathematics. In particular, Boolean Algebra has already become an indispensable tool to the computer designer and, no doubt, has benefitted from the practical interest of the engineer. Modern algebra as a whole is a source of ideas which may be applied to the design of computers.

To achieve this objective in a reasonable space, it was necessary to select a suitable topic from the very abundant material on digital computers. The topic which conveys most of the ideas essential to the understanding of computers and is also amenable to mathematical description, is that of structural organisation. The word "structural" is here used in a specialised sense. The study of digital computers can be divided into five general areas, viz.,

1. Circuits and Components
2. Logical Design

3. Structural Organisation
4. Programming (including automatic programming languages and numerical analysis.)
5. Theoretical studies of sequential machines, Turing machines, etc.

The design of a computing system would involve the first four at least, with a considerable degree of co-operation among them. In practice, the divisions are not clearly defined as there is much overlapping between adjacent areas. First of all, a programming language would be developed, taking into consideration the potential applications of the machine. The structural designer would then consider how this language might be implemented by transfers of data between gross elements of the machine such as arithmetic registers, memory registers and input-output registers, with a view to economising in some sense; e.g., subtraction can usually be accomplished with virtually the same transfers as addition. The logical designer translates the resulting structure of the machine into greater detail using such tools as Boolean algebra, coding theory (number representation systems) and switching circuit theory. Finally the circuit designer translates a Boolean description of the machine into hardware. Many iterations of this entire

process are required to design a system. In the early days of computers, the process operated in reverse with the programmer taking whatever the circuit designers could provide and making the best of an awkward programming language.

The description of computers at the level of structural organisation has two main advantages from the viewpoint of this thesis. First of all, it enables us to suppress as much of the detailed logical and circuit design as desired and, at the same time, avoid the relatively gross programming languages which tend to obscure the basic principles of operation of the machines. Secondly, the structural organisation of computers is amenable to mathematical description in terms of Boolean concepts. It was felt that a study of computers in general at this level would be less revealing than a detailed description of one particular computer currently in use.

The IBM 1620 was selected for study not because it is representative of current digital computers in general, but because of its unusual sequential structure, whose study suggests at least two fields of current theoretical interest. The first, that of deterministic (sequential) machines is covered in Chapter 6 and the second is simply

that of finding a language which will describe precisely such a complex sequential process as the operation of a digital computer and describe it in a form satisfactory to specialists in the five areas mentioned above. The only language known to the writer which even remotely approaches this problem is the algorithmic language described by K. E. Iverson+[1]. The language is used to present the detailed structure of the 1620 in Chapter 5. The language might also have been used in Chapter 4 to shorten the description of encoding and decoding methods but, since conventional mathematical notation provides an adequate description, it was decided not to burden the chapter with two notations. An example of the brevity the notation permits in this field may be found in Iverson [2]. A subset of the language is employed in Chapter 5 but Iverson [1],[2],[3], shows that the full language may be applied to the wide range of problems where finite sequential processes are encountered. The language is here used with the deliberate suggestion that it, or something like it, may be the language to which Bartee referred.

We have proceeded thus far without making any overt definition of what constitutes a digital computer. The reason is simple; the digital computer is evolving so

+ The names of authors will be used as a reference system. When an author has several papers referred to in this thesis the name will be followed by the number of the paper.

rapidly that a precise definition of a computer is likely to appear somewhat narrow in the light of new developments. We shall conclude this introductory chapter with a short discussion of machines in general. The computers which have received so much attention in recent years form a special class of machines in general and may be distinguished from the others by giving them their full title, viz., general-purpose, stored-program, electronic, digital machines. The word "machine" will be left undefined as a full discussion leads rapidly into philosophy and (usually) obscurity. The attributes of a machine are suggested by the word "automaton." So let us avoid the problem by calling a machine an automaton (undefined) which handles numbers or information (undefined, see Chapter 3).

Digital machines represent information in discrete as opposed to continuous fashion. Information is transferred inside the machine electronically rather than by other physical means such as the rotation of a mechanical gear-wheel. The operation of the machine is controlled by an <u>internally</u> stored list of instructions (program) which is accessible to the machine so that it may modify its list of instructions. A general-purpose machine is designed to handle a wide class of problems. This means that its program can be changed easily from outside the machine in

contrast to the special-purpose machine whose program is not readily accessible from the outside although it may be easily modified by the machine itself.

The adjectives "general-purpose," "stored-program," "electronic" and "digital" all suggest variants which may or may not have been realized as useful machines. However, we shall be concerned in this thesis only with the very narrow class of machines described by these properties. These particular machines are important because of their notable success as practical calculating tools in many diverse areas of application.

# CHAPTER 2

# Codes in Common Use

## 2.1 Introduction

The decimal symbols, 0,1,...,9 constitute an important means of representing information and performing arithmetic operations. The decimal system has become so firmly embedded in human usage that when an alternative system is proposed, its proponents must have some very strong reasons for preferring it.

Such proposals seldom have more than minor advantages over the decimal system but there is one very good reason for using the binary system - it is an indispensable tool in the theory and design of electronic devices, such as digital computers. The binary system is not proposed as a replacement for the decimal system but as a supplement. A more useful replacement - from a designer's point of view - for the decimal system would be the octal or the hexadecimal systems which have as bases powers of 2. The practical development of digital computers has depended largely on the reliability offered by bistable (binary) devices. Extreme reliability is essential when one considers that something like $10^6$ such devices are required for a reasonable computer. Further, the programming

of a computer is primarily sequential, involving perhaps $10^5$ steps for a reasonable algorithm. A "step" might use from 100 to 1000 bistable devices at a time. Under these conditions even a small decrease in reliability would lead to an enormous increase in the amount of checking required. Let us note, in passing, the magnitude of this engineering accomplishment in reliability. The user of a modern electronic computer can reasonably expect his machine to operate for days or weeks without an error. The time-scale itself is not impressive until we consider that in 24 hours the computer could carry out about $10^9$ steps such as adding two ten-digit decimal numbers together.

Another reason for choosing binary is that a digital computer is first a logical machine and secondly an arithmetical machine. Mathematical logic is fundamentally binary and is not efficiently represented in decimal symbols.

Clearly, then, the designer of digital computers is caught between the universal usage of decimal symbols and the engineering necessity of binary symbols, and has, in addition, the problem of implementing logical operations efficiently. The various elementary responses to this dilemma are the subject of this chapter and are, as it happens, the present state-of-the art as far as commercially available computers are concerned.

This chapter discusses in detail ways of representing decimal digits as sets of binary digits, the so-called binary-coded-decimal codes. The ideas introduced are important in the three succeeding chapters, usually as important special cases of more general concepts. Information theory (Chapter 3) deals with finite alphabets like (0,1) and (0,1,...,9) for the transmission of information and develops concrete notions of efficiency and other properties of such alphabets. Binary-coded-decimal codes have certain properties which can be used to detect and correct errors which occur when they are in transit from place to place within the machine - the subject of Chapter 4. Some of the problems of implementing decimal arithmetic using binary devices form a substantial part of this chapter as a preliminary to Chapter 5 which describes the IBM 1620, a binary-coded-decimal computer. These problems belong to a significant new area of research which is an offshoot of the simple coding problem and is concerned primarily with new ways of performing fast arithmetic. Some of the more promising theoretical suggestions abandon binary representations in favour of residue systems (Garner, [1]).

## Section 2.2 Bistable Devices and Boolean Notation

The bistable devices used in a digital computers

are based on some physical phenomenon, usually electrical or magnetic. The exact nature is of no concern to us here. The important properties are 1) that the device has two distinguishable states and 2) that it can be set in either state and retain that state indefinitely until changed.

We can arbitrarly assign a name to each state, say ON or OFF, a or b, or 0 or 1, true or false. If we choose 0 and 1 as symbols, we obviously have the basis for both Boolean manipulations and binary representation of numbers. The use of Boolean algebra in designing switching circuits was proposed by C. E. Shannon in 1938 and since then has become an indispensable tool for understanding and designing computers. Binary arithmetic is performed by a sequence of Boolean operations such as +, ., complement, which are described by the tables

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| . | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

$0' = 1$

$1' = 0$

Table 2.2.1

To add two binary digits, x,y we need to generate the corresponding sum digit(s) and a possible carry digit (c). Boolean expressions for s and c are

$$s = x.y' + x'.y$$

$$c = x.y$$

When large numbers of bistable devices are used and the logic becomes very complex, Boolean algebra is the only useful method of analysis. A typical problem is that of minimizing the number of devices and basic operations required to implement a function such as binary arithmetic.

Computers which use pure binary arithmetic are probably the most common type because of the relative ease of implementation.

## Section 2.3 Number Representation Systems and Codes

Many users of computers would prefer to deal with the familiar decimal number system and it is possible to make the binary devices operate in a "quasi-decimal" fashion, at the cost of increased logical complexity. Most machines are organised around the manipulation of numbers expressed in ordinary positional notation, i.e., the sequence of digits $b_1b_2...b_n$ is taken to mean the natural number

$$\sum_{i=1}^{n} b_i 2^{n-i}, \text{ where } b_i = 0 \text{ or } 1.$$

The user of the machine is often faced with making the translation into decimal positional notation, $d_1 d_2 \ldots d_m$ which represents

$$\sum_{i=1}^{m} d_i 10^{m-i}, \text{ where } d_i = 0,1,\ldots,9.$$

The translation from binary to decimal and back is an arithmetic one and, though straightforward, is somewhat tedious. We can avoid this problem to a great extent by using binary-coded decimal notation. We arbitrarily assign a <u>set</u> of binary digits to represent a decimal digit. The most common way of doing this is to associate 4 binary digits with each decimal as follows

| Decimal | | Binary |
|---|---|---|
| 0 | - | 0000 |
| 1 | - | 0001 |
| 2 | - | 0010 |
| 3 | - | 0011 |
| 4 | - | 0100 |
| 5 | - | 0101 |
| 6 | - | 0110 |
| 7 | - | 0111 |
| 8 | - | 1000 |
| 9 | - | 1001 |

Table 2.3.1

The decimal number 33 would appear as 100001 in true binary and as 00110011 in this binary-coded decimal.

Unfortunately, arithmetic operations cannot be performed directly in this representation. We will return to this point later.

The code we have cited is called the 8,4,2,1 code because of the weight assigned to each position. Many other codes are possible - indeed, 16!/6! of them. However, according to Weeg there are only 88 different weighted codes of which 17 have all positive weights. Weeg also notes that the 8,4,2,1 code is the only positively weighted code which represents each decimal digit uniquely. The others permit choices of representation. For example, the 7,4,2,1 code permits 1000 or 0111 as the representation of digit 7.

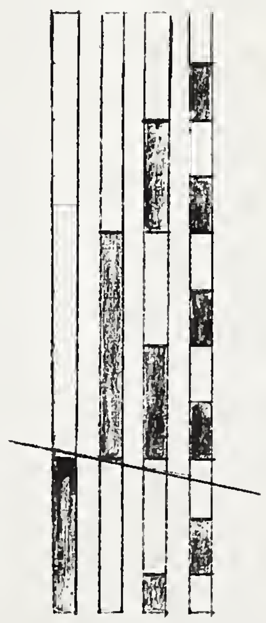
The remaining non-weighted codes have not been used much as the circuits associated with non-weighted codes are somewhat more complicated than those for weighted codes. One class of non-weighted codes has a special application in digitalizing continuous physical measurements such as the angular position of a rotating shaft. These

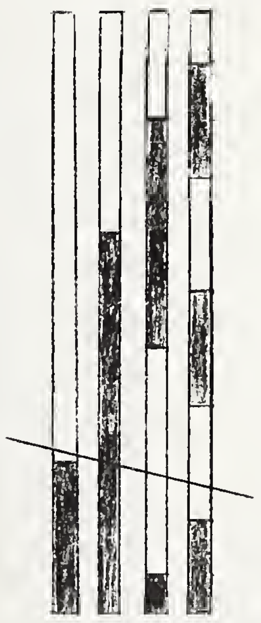
are the Gray codes or reflected binary. An example is

Table 2.3.2

| | |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0011 |
| 3 | 0010 |
| 4 | 0110 |
| 5 | 0111 |
| 6 | 0101 |
| 7 | 0100 |
| 8 | 1100 |
| 9 | 1101 |
| 10 | 1111 |
| 11 | 1110 |

The useful property of these codes is that only one binary digit at a time is changed in passing from, say 5 to 6. This is made clear from a diagram





| 8,4,2,1 | | Gray |
|---|---|---|
| 0 0 0 0 | 0 | 0 0 0 0 |
| 0 0 0 1 | 1 | 0 0 0 1 |
| 0 0 1 0 | 2 | 0 0 1 1 |
| 0 0 1 1 | 3 | 0 0 1 0 |
| 0 1 0 0 | 4 | 0 1 1 0 |
| 0 1 0 1 | 5 | 0 1 1 1 |
| 0 1 1 0 | 6 | 0 1 0 1 |
| 0 1 1 1 | 7 | 0 1 0 0 |
| 0 0 0 0 | 8 | 1 1 0 0 |
| 1 0 0 1 | 9 | 1 1 0 1 |
| 1 0 1 0 | 10 | 1 1 1 1 |
| ⋮ | 11 | ⋮ |

Table 2.3.3

As indicated, a slight error in alignment of the "reading" mechanism (the slanted line) in the case of the 8,4,2,1 code gives a reading of 0 0 0 0 when it should be 0 1 1 1(7) or 1 0 0 0(8). With the Gray code the maximum error possible is one. In general, an n binary digit Gray code will limit the magnitude of an error to $1/2^n$, where the 8,4,2,1 code might permit a error of $2^n$. Tompkins presents an extremely detailed study of Gray codes.

We can dismiss non-weighted codes as internal representations of decimal digits because of the difficulty in performing arithmetic. Arithmetic is not the only consideration, of course, but it is certainly very important in computer design. To exemplify a consideration of a different kind, let us look at the 7,4,2,1 code,

| | |
|---|---|
| 0 | 0 0 0 0 |
| 1 | 0 0 0 1 |
| 2 | 0 0 1 0 |
| 3 | 0 0 1 1 |
| 4 | 0 1 0 0 |
| 5 | 0 1 0 1 |
| 6 | 0 1 1 0 |
| 7 | 1 0 0 0 |
| 8 | 1 0 0 1 |
| 9 | 1 0 1 0 |

Table 2.3.4

we note that no digit has more than two 1's. If the computer uses bistable devices in which 1 means ON and 0 means OFF then the power dissipation would be minimised. Other codes such as the 4,3,1,1 code would have up to twice this power dissipation. Admittedly, this is a minor problem but it serves to indicate that the choice of representation in physical devices is by no means trivial.

## <u>Section 2.4 Arithmetic in Binary-Coded Decimal Notation</u>

Let us now examine some of the problems of performing addition in binary-coded decimal. We will assume that true binary arithmetic is easily realised in terms of bistable devices, i.e., if we have two binary digits $x_n, y_n$ and a binary carry digit $c_{n-1}$ from the previous operation we can construct the following table of eight possible states.

| $x_n$ | $y_n$ | $c_{n-1}$ | $s_n$ | $c_n$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Table 2.4.1

In the 8,4,2,1 code binary carries are handled naturally from the above table within each 4-binary-digit group. A carry from one group into the next must occur if the sum exceeds 1001, and the sum digit must be reduced modulo 1010, e.g.,

| | | | | | |
|---|---|---|---|---|---|
| 1 5 6 | | 0001 | 0101 | 0110 | |
| 4 8 2 | | 0100 | 1000 | 0010 | |
| 6 3 8 | | 0110 | 0011 | 1000 | |
| 1 0 | Carries (Decimal) | 0000 | 0000 | 1100 | Binary Carries |
| | | 1 | 0 | | Decimal Carries |

Fig. 2.4.1

Clearly the machine can use the same logic for each position within the 4-digit groups. This is not true for the other positive-weighted codes where some intra-group logic would be required.

Decimal carries may be obtained automatically if we use a modified form of the 8,4,2,1 code called the "excess-3" code.

| | |
|---|---|
| 0 | 0 0 1 1 |
| 1 | 0 1 0 0 |
| 2 | 0 1 0 1 |
| 3 | 0 1 1 0 |
| 4 | 0 1 1 1 |
| 5 | 1 0 0 0 |
| 6 | 1 0 0 1 |
| 7 | 1 0 1 0 |
| 8 | 1 0 1 1 |
| 9 | 1 1 0 0 |

Table 2.4.4

Decimal arithmetic may be performed on numbers coded in this representation by first adding them as if they were pure binary numbers and noting the carries which occurred from one decimal position to another, (every fourth binary digit). The sum digits in the position which caused a carry have to be adjusted by +3 and the others by -3.

However, this operation is easily accomplished by binary arithmetic and hence requires little additional complication. e.g.,

| | | | | |
|---|---|---|---|---|
| 1 5 6 | 0100 | 1000 | 1001 | |
| 4 8 2 | 0111 | 1011 | 0101 | |
| 6 3 8 | 1100 | 0011 | 1110 | Binary Sum |
| | -0011 | +0011 | -0011 | Corrections. |
| | 1001 | 0110 | 1011 | Adjusted Sum |

Fig. 2.4.2

This property alone would not recommend the "excess-3" code for arithmetic. It has also the useful property that its digits may be complemented, digit by digit, to produce the nines-complement of each decimal digit e.g., if we replace 1's by 0's and 0's by 1's in 1001 (6) we obtain 0110 (3). We can then use the adder to perform subtraction at little extra cost. Some of the positive-weighted codes permit this type of complementation (one of the easier manipulations with binary devices.) A necessary (and sufficient) condition for this to be possible with a weighted code is that the sum of the weights should be 9 and the representations of each decimal digit and its nines-complement be symmetrically situated with respect to the center of a table of combinations of 4 binary digits.

| | 4 3 1 1 | |
|---|---|---|
| 0 | 0 0 0 0 | |
| 1 | 0 0 0 1 | |
| (1) | 0 0 1 0 | |
| 2 | 0 0 1 1 | |
| 3 | 0 1 0 0 | |
| (4) | 0 1 0 1 | |
| (4) | 0 1 1 0 | |
| 4 | 1 0 0 0 | ...Axis of symmetry |
| 5 | 0 1 1 1 | |
| (5) | 1 0 0 1 | |
| (5) | 1 0 1 0 | |
| 6 | 1 0 1 1 | |
| 7 | 1 1 0 0 | |
| (8) | 1 1 0 1 | |
| 8 | 1 1 1 0 | |
| 9 | 1 1 1 1 | |

Table 2.4.3

The representations of 1,4,5,8 in the 4,3,1,1 code permit several choices some of which satisfy the symmetry requirement.

Unfortunately, the popular 8,4,2,1 code does not permit this simple type of binary complementation. However since complementing the representations of decimal digits in the 8,4,2,1 code is considerably simpler than building a complete subtracter, complementary arithmetic is often used with this code.

Let us review the rules for arithmetic in

complementary decimal form and then examine their implementation in binary-coded decimal.

To complement a number, replace the least-significant non-zero digit by its tens-complement; each remaining digit to the left of the first complemented digit is replaced by its nines-complement.

<u>Rules for Addition</u>:

1. Complement negative numbers.
2. Add the numbers.
3. Recomplement the sum if it is negative.

<u>Rules for Subtraction</u>:

1. Complement negative numbers.
2. Complement the subtrahend and add.
3. Recomplement the sum if negative.

| | 8 4 2 1 | Nines Complement | Tens Complement |
|---|---|---|---|
| 0 | 0 0 0 0 | 1001 | 0000 |
| 1 | 0 0 0 1 | 1000 | 1001 |
| 2 | 0 0 1 0 | 0111 | 1000 |
| 3 | 0 0 1 1 | 0110 | 0111 |
| 4 | 0 1 0 0 | 0101 | 0110 |
| 5 | 0 1 0 1 | 0100 | 0101 |
| 6 | 0 1 1 0 | 0011 | 0100 |
| 7 | 0 1 1 1 | 0010 | 0011 |
| 8 | 1 0 0 0 | 0001 | 0010 |
| 9 | 1 0 0 1 | 0000 | 0001 |

Table 2.4.4

<u>Ex.</u> 158 + (-422) = -264

```
     158
  +  578  Complement of -422
     736  ⟶ -264 (Recomplement)
```

|  |  |  |  |  |
|---|---|---|---|---|
| 158 |  | 0001 | 0101 | 1000 |
| + 578 | (Complement) | 0101 | 0111 | 1000 |
| 736 | (Sum) | 0111 | 0011 | 0110 |
| - 264 | (Recomplement) | -0010 | 0110 | 0100 |

Fig. 2.4.3

An adder based on this scheme would be able to add or subtract positive and negative numbers. The additional functions would be some simple sign logic to control complementation and the nines- and tens- complements of the decimal digits. This is only one of several possible adding methods but is perhaps the simplest to implement in bistable devices. Complementary arithmetic is certainly the most common in both binary and binary-coded decimal computers.

At this point we should summarize the present situation in regard to 4 digit binary-coded decimal representations. The 8,4,2,1 code is by far the most common in use except for true binary. It presents a fairly natural conversion between binary and decimal without arithmetic. It is a weighted code and is reasonably

convenient for arithmetic. Other codes have lesser advantages to the computer designer - none of them, however, outweigh the convenience to the computer user of a familiar notation.

## Section 2.5 Error-Detecting Codes

Until now we have made the assumption that 4 binary digits per decimal digit are sufficient. Certainly we need at least four, but five or more binary digits have been used as well. The use of four binary digits per decimal digit already results in a "waste" of some of the possible combinations. A binary-coded decimal number requires 4n binary digits to represent $10^n$ different numbers. In true binary $10^n$ numbers can be represented by $\log_2 10^n =$ 3.32n binary digits. The use of more than 4 digits per decimal obviously makes this worse.

The "wasted" space in these codes can be put to use in checking for errors in a machine. For example, if the combinations 1010,1011,...,1111 turn up inside a machine using the 8,4,2,1 code then obviously an error has occurred and the machine can signal to the operator that something is wrong. However, checking for forbidden combinations throughout the machine is quite complicated - and inadequate; simpler methods are available.

Consider a "2-out-of-5" code below.

| | |
|---|---|
| 0 | 01100 |
| 1 | 10001 |
| 2 | 10010 |
| 3 | 00011 |
| 4 | 10100 |
| 5 | 00101 |
| 6 | 00110 |
| 7 | 11000 |
| 8 | 01001 |
| 9 | 01010 |

Table 2.5.1

There are exactly 2 1's in each representation and a single error would change a 1 to a 0 or a 0 to a 1. The machine simply checks for the presence of 2 1's and signals an error for fewer or more 1's. This code also detects some double, triple, etc, errors. The "2-out-of-5" code is not a weighted code but weighted codes, such as the bi-quinary, are available.

| | 5 | 0 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 7 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

Table 2.5.2

The inefficiency of this code is obvious. Fortunately, other more powerful error-detection schemes are available. They are examined in Chapter 4 since we need a more general viewpoint from which to survey them. Here, it will be sufficient to indicate the elementary considerations from which they originated. The schemes are based on the idea of "parity checks." Let us add one binary digit to the 8,4,2,1 code and use it to make the number of 1's in each representation even.

| | 8 | 4 | 2 | 1 | (0) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 1 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 0 |

Table 2.5.3

In other words, the sum modulo 2 of the binary digits of each representation is zero. This type of checking is easily accomplished in a binary device. Clearly, we could have used an odd check instead. The parity checks may operate over selected digits of a representation and

[illegible]

several parity checks included in a representation will permit not only error-detecting but error-detecting as well. Chapter 4 will consider error-correcting from a more general viewpoint.

Many computers use single parity checks which permit detection of odd numbers of errors. A few computers use error-correcting but, so far, it has been cheaper to design a system conservatively than to include relatively expensive error-correcting circuits.

# CHAPTER 3

# An Introduction to Information Theory

## Section 3.1 Introduction: Elementary Notions of Information.

Digital computers are frequently called "information-processing" devices. This is true in a fairly wide sense of the word "information" but, more important, it is true in the sense of the mathematical definition of information. Mathematical information theory is based on a very few assumptions about the statistical nature of information and has nothing to say about the value of the information itself to a potential receiver. A digital computer can be regarded as a collection of devices which transmit information from place to place within the computer with varying probabilities of success. Indeed we shall present in Chapter 5 a digital computer which even performs arithmetic by transmitting rather than manipulating information. The present position is that, while information theory is not yet in the "need-to know" category, the designer (and the user) of digital computers will soon have to be aware of its results.

Like many words which are used as names for mathematical concepts, "information" has many different

meanings in common usage. In particular, information is associated with the words "knowledge" and "meaning" and the idea of a mathematical theory of information or knowledge is so attractive that it has led to many over-enthusiastic misapplications of the present information theory. Little progress was made until the concept of information was stripped of all connotations of "meaning." The first steps in this direction were taken by Hartley who was interested in the transmission of information from an engineering, rather than a philosophical, point of view. Some of the statistical properties of information were known to communication engineers at that time and Hartley's contribution was to suggest that information might be regarded as instructions to select one event from a number of events which could possibly occur in a given situation. To take a simple example, suppose it was desired to convey the information that one of eight events $E_1$, $E_2, \ldots, E_8$ has occurred and the only means available is a device which can transmit the binary digits 0 and 1. If the sender of the information and the receiver agree that the digit 1 represents a "yes" answer and the digit 0 a "no" answer and that a sequence of questions be asked, viz. 1) "Is the event which occurred in the first half of the ordered set $E_1, E_2, \ldots, E_8$?", 2) "Is the event in the

first half of the subset defined by the answer to 1)?" and 3) "Is the event in the first half of the subset defined by the answer to 2)." Then if the event happens to be $E_5$, the answers to the three questions are no, yes, yes and the sequence of binary digits which should be sent is 0,1,1. Table 3.1.1 gives the binary sequences for all eight events.

| Event | 1st | 2nd | 3rd | Questions |
|---|---|---|---|---|
| $E_1$ | 1 | 1 | 1 | |
| $E_2$ | 1 | 1 | 0 | |
| $E_3$ | 1 | 0 | 1 | |
| $E_4$ | 1 | 0 | 0 | |
| $E_5$ | 0 | 1 | 1 | |
| $E_6$ | 0 | 1 | 0 | |
| $E_7$ | 0 | 0 | 1 | |
| $E_8$ | 0 | 0 | 0 | |

Table 3.1.1

It is clear from the table that each event $E_i$ is associated with a unique sequence of binary digits. The process of associating an event with a sequence of digits (not necessarily binary) is called encoding and is a highly developed topic of information theory (see

Chapter 4). In general, encoding need not proceed on the logical basis of Table 3.1.1; indeed any assignment of unique 3-digit sequences would suffice if the sequence can be examined as a whole. The encoding system could be used for the transmission of the octal digits 0,1,2,...,7 to a digital computer which uses binary digits internally. The event $E_i$ in that case would be the occurrence of the octal digit i-1 in, say, an octal number which is to be inserted in the memory of the computer. The sets of digits or symbols {0,1}, {0,1,2,...,7} are called alphabets, by analogy with the 26 symbol English alphabet A,B,C,...,Z. The binary alphabet is very important because it is the alphabet of the reliable electronic devices used in communication systems. It is also the simplest alphabet possible since an alphabet of one symbol can convey no information, as Lewis Carroll* pointed out.

It is a well-known fact that the English alphabet has a definite statistical structure, (see Appendix 3). This fact is especially important to communication engineers in making efficient use of telegraph and other communication channels. In his code

* It is a very inconvenient habit of kittens (Alice had once made the remark) that, whatever you say to them, they always purr. "If they would only purr for 'yes,' and mew for 'no' or any rule of that sort," she had said, "so that one could keep up a conversation! But how <u>can</u> you talk with a person if they always say the same thing?"

<u>Through the Looking Glass</u>.

Morse took advantage of the statistical structure of English by assigning short code sequences to the most frequent letters and long sequences to the infrequent letters. This can be expressed more formally as the minimisation of

$$\sum_{i=1}^{n} p(E_i)C(E_i) \ldots .3.1.2$$

where $p(E_i)$ is the probability that the event $E_i$ will occur and $C(E_i)$ is the "cost" of transmitting the information used to specify the occurrence of $E_i$. Assuming that the costs of transmitting the binary digits 0 and 1 are the same, then 3.1.2 may be expressed as

$$\sum_{i=1}^{n} p(E_i)L(E_i) \ldots \quad 3.1.3$$

where $L(E_i)$ is the number of binary digits used to specify the event $E_i$. A method of finding the encoding which minimises 3.1.3 was discovered by Huffman [2] and is discussed in Appendix 3.

The real impetus for developing a statistical theory of information did not come from the probabilistic nature of information itself but from the observation that the physical channels which were used to transmit information were liable to error and that this behavior

could be described statistically. For example, the only properties of a binary transmission channel that are important in this sense are that 1) there is a probability q of receiving a 0 when a 1 is sent and a probability 1-q of receiving a 1 when a 1 is sent and 2) there is a probability p that a 1 is received when a 0 is sent and a probability 1-p that a 0 is received when a 0 is sent. More compactly

$$
\begin{aligned}
p(1|1) &= 1-q \\
p(0|1) &= q \qquad\qquad 0 \leq p,q < 1/2 \\
p(0|0) &= 1-p \\
p(1|0) &= p
\end{aligned}
$$

where $p(y|x)$ is the conditional probability that y is received when x is sent. In a useful channel, p and q would be very much smaller than 1. Information theory is based on this and similar simple models of communication channels, and has contributed a great deal to the understanding of how real communication channels work. The first major achievement of the theory was that information could be encoded so that it would be transmitted perfectly through channels which were liable to error (Shannon [1]).

## Section 3.2 Mathematical Theory of Information

The modern theory began in 1948 with Shannon's work and since then has been put in rigorous mathematical form by many mathematicians, notably Feinstein, Khinchin and, recently, Wolfowitz. The first task of the theory was to find a suitable measure for the amount of information, based on a few assumptions about the statistical nature of the transmission of information. A typical starting point would be that information is produced (somehow) and transmitted (somehow) to a receiver. The receiver could be said to acquire information when it is informed of the occurrence of an event whose occurrence was not previously certain. Furthermore, the more improbable the event, the more information is conveyed by knowledge of its occurrence. Let $I_x$ be the amount of information conveyed to the receiver by the knowledge of the occurrence of an event x. Since x is specified completely by its probability $p_x$, we can assume that $I_x = I(p_x)$. Also $I_x$ is non-negative. Consider the case where the information to be transmitted is the occurrence of x or the non-occurrence of x. Since these two events are mutually exclusive, $p_x+p_{x'}= 1$ ($x'$ the non-occurrence of x) and $I_{x'} = I(p_{x'})$. Since the probability of receiving the amount of information $I(p_{x'})$ is $p_{x'}$ then the

average amount of information receivable is $H(x,x') = H(p_x,p_{x'}) = p_x I(p_x)+p_{x'} I(p_{x'})$. This is easily extended to n mutually exclusive events $x_i, i=1,\ldots,n$.

$$H(x_1,x_2,\ldots,x_n) = H(p_{x_1},\ldots p_{x_n}) = p_{x_1} I(p_{x_1})+\ldots+ p_{x_n} I(p_{x_n})\ldots(1)$$

$$= \sum_{i=1}^{n} p_{x_i} I(p_{x_i}).$$

If a $p_{x_i}$ happens to be zero, it is natural to drop the corresponding term from the expression for $H(x_1,\ldots,x_n)$, as an event with zero probability can convey no information. (H, the average amount of information is often called "entropy" because of its close connection with the physical concept.)

## Section 3.3 Properties of the Entropy Function

The form of the H function can be determined from the following four assumptions - indeed over-determined as it has been shown that certain choices of three of them are sufficient.

1. Continuity: A slight change in the probabilities $p(x_i)$ should not result in a large change in H.

i.e., $H(x_1,\ldots,x_n) = H(p(x_1),\ldots p(x_n))$ is continuous in $p(x_k), k=1,\ldots,n, 0\leq p(x_k)\leq 1$.

2. Symmetry: The order of the events $x_i$ is unimportant. $H(p(x_1),p(x_2),\ldots p(x_n))=H(p(x_2), p(x_1),\ldots,p(x_n))$.

3. Extremal Property: When all events $x_i$ are equally likely a maximum amount of information is received with the knowledge of which event occurred.

$$\max H(p(x_1),p(x_2),\ldots,p(x_n) = H(\tfrac{1}{n},\ldots,\tfrac{1}{n})$$

4. Additivity: Suppose the event $x_n$ is a composite event consisting of m mutually exclusive events $u_1,u_2,\ldots,u_m$ with associated probabilities $q_1,q_2,\ldots,q_m$ and $p(x_n) = \sum_{k=1}^{m} q_k$, then since

$$\frac{q_1}{p(x_n)} + \frac{q_2}{p(x_n)} +\ldots+ \frac{q_m}{p(x_n)} = 1,$$ the event $x_n$, when its occurrence is certain, may be regarded as a probability scheme with an entropy function $H(\frac{q_1}{p(x_n)}, \frac{q_2}{p(x_n)},\ldots, \frac{q_m}{p(x_n)})$.

We now have three probability schemes whose H functions are related in a linear fashion, viz.

$$H(p_1,p_2,\ldots,p_{n-1},q_1,q_2,\ldots,q_m)$$

$$= H(p_1,p_2,\ldots,p_n)+p_nH(\frac{q_1}{p_n}, \frac{q_2}{p_n},\ldots,\frac{q_m}{p_n})$$

where $p_i = p(x_i)$.

These assumptions led Shannon to the form

$$H(p_1,p_2,\ldots,p_n) = C\sum_{k=1}^{n} p_k\log_2 p_k$$

where $C > 0$, $\sum_{k=1}^{n} p_k = 1$.

The choice of the base 2 for the logarithm is suggested by the needs of the communication engineer and is fairly standard. This leads to the unit of information being defined as a "bit."

$$H(p_1,\ldots,p_n) = -\sum_{k=1}^{n} p_k\log_2 p_k {}^{+} \text{ bits} \quad \ldots 3.3.1$$

where one bit is the amount of information conveyed by the selection of one of two equally likely events, $H(\frac{1}{2}, \frac{1}{2}) = 1$.

Obviously, this form of H satisfies Property 2, and Property 1 since the logarithmic function is

+ In the sequel we drop the suffix 2 from $\log_2 x$.

continuous in (0,1] and since $\lim_{x\to 0} (x \log x) = 0$.

Property 3 asserts that

$$H(p) = H(p(x_1),p(x_2),\ldots,p(x_n)) \leq H\left(\frac{1}{n},\frac{1}{n},\ldots,\frac{1}{n}\right).$$

By 3.3.1,

$$H\left(\frac{1}{n},\frac{1}{n},\ldots,\frac{1}{n}\right) = -n\left(\frac{1}{n}\log \frac{1}{n}\right) = \log n.$$

Hence we have to show that $H(p) \leq \log n$. This can be done with the aid of the following lemma.

<u>Lemma</u>. $\ln x \leq x-1$.

Since ln x is a convex function for $x>0$, ln x lies below the tangent to ln x at $x = 1$. The equation of this tangent is given by

$$y = \left(\frac{d(\ln x)}{dx}\bigg|_{x=1}\right) (x-1)$$

$$= x - 1.$$

Hence $\ln x \leq x - 1$

and $\log x = \ln x \log_2 e \leq (x-1) \log e$.

Now,

$$H(p) - \log n = \sum_{i=1}^{n} p_i \log(\frac{1}{p_i}) + \log \frac{1}{n}$$

$$= \sum_{i=1}^{n} p_i \log(\frac{1}{p_i}) + \sum_{i=1}^{n} p_i \log \frac{1}{n}$$

$$= \sum_{i=1}^{n} p_i \log (\frac{1}{p_i n})$$

$$\leq \sum_{i=1}^{n} p_i (\frac{1}{p_i n} - 1) \log e \text{ (lemma)}$$

$$\leq \log e \{ \sum_{i=1}^{n} (\frac{1}{n} - p_i) \}$$

$$\leq 0.$$

Hence $H(p(x_1),p(x_2),\ldots p(x_n)) \leq \log n$ and Property 3 is satisfied.

Property 4 may be verified by repeated use of Eqn. 3.3.1.

$$H(p_1,p_2,\ldots,p_{n-1},q_1,q_2,\ldots,q_m)$$

$$= -\sum_{i=1}^{n-1} p_i \log p_i - \sum_{i=1}^{m} q_i \log q_i$$

$$= -\sum_{i=1}^{n} p_i \log p_i + p_n \log p_n - \sum_{i=1}^{m} q_i \log q_i$$

$$= H(p_1,p_2,\ldots,p_n) + p_n \log p_n - \sum_{i=1}^{m} q_i \log q_i$$

$$= H(p_1,\ldots,p_n) + p_n \sum_{i=1}^{m} \frac{q_i}{p_n} \log p_n - \sum_{i=1}^{m} q_i \log q_i$$

$$= H(p_1,\ldots,p_n) - p_n \sum_{i=1}^{m} \frac{q_i}{p_n} \log \left(\frac{q_i}{p_n}\right)$$

$$= H(p_1,\ldots,p_n) + p_n\, H\left(\frac{q_1}{p_n},\frac{q_2}{p_n},\ldots,\frac{q_m}{p_n}\right)$$

The special case $n = 2$ merits attention as the "language" of computers and electrical communication in general is based on the selection of one of two states of a device. We have $p_1+p_2 = p(x_1)+p(x_2) = 1$

$$\text{and} \quad H(p_1,p_2) = -p_1\log p_1 - p_2\log p_2$$

$$= -\, p_1\log p_1 - (1-p_1)\log(1-p_1)$$

A plot of $H(p_1,1-p_1)$ versus $p_1$ reveals the expected maximum at $p_1=1/2$.

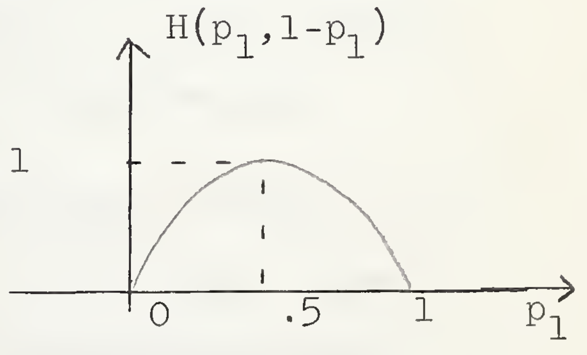


Fig. 3.3.1

In this connection we should point out the difference between the term "binary digit" and "bit" as they are often used interchangeably. "Binary digits" are the letters of the binary alphabet just as A,B,...,Z are the letters of the English alphabet. Binary digits may or may not contain information. A "bit" is the unit of the amount of information in whatever form it may be presented. For example, if we have a series of binary digits "passing through" a device which can be in one state, <u>called</u> 0, or the other state, <u>called</u> 1, then we could observe over a long period of time the frequency of the state 0 and the state 1. We could establish the probabilities that the device would be in state 0 and state 1, say $p_1$ and $p_2$ respectively. If $p_1 = p_2 = \frac{1}{2}$ then $H(p_1,p_2) = 1$ and one binary digit would convey 1 bit of information. On the other hand, if $p_1 \neq p_2$ then $0 \leq H(p_1,p_2) < 1$; in which case one binary digit would convey less than one bit of information. Clearly, some care is necessary in the use of the word "bit". In this thesis, "bit" will be used to mean the unit of information.

We have postulated a measure $H(p)$ for the average amount of information and it has been shown that this measure is well-defined and has convenient properties which accord well with intuitive

notions. H(p), may be taken as the definition of average amount of information conveyed by the selection of one of the possible events of a finite discrete probability scheme consisting of a set of mutually exclusive events $(x_1, \ldots, x_n)$ and the associated probabilities $(p(x_1), \ldots, p(x_n))$. The only important properties of the events are their probabilities of occurrence and that they are mutually exclusive. The probability scheme can obviously be generalised to infinite and continuous schemes but those are beyond the scope of our interest at the moment, besides being considerably more difficult to present.

## Section 3.4 Communication Systems

The generalisation which is of interest, that is, to two-dimensional finite discrete schemes, provides us with a mathematical model of a communication system. Let us first introduce more concrete ideas of what constitutes a communication system. The necessary elements are 1) a source of information, 2) a channel for conveying the information and, 3) a receiver. The following "black-box" model illustrates the connections of the system.

$$\begin{tikzcd} \boxed{\text{SOURCE}} \arrow[r] & \boxed{\text{CHANNEL}} \arrow[r] & \boxed{\text{RECEIVER}} \end{tikzcd}$$

Fig. 3.4.1

The source has an "alphabet" consisting of a finite number of letters (A,B,...,Z; or 0,1; or 0,1...,9 etc.) each with an associated probability of being selected. The letters (or events) are passed to the channel for transmission to the receiver which also has an alphabet. The receiver's alphabet may differ from that of the source, in which case, the channel would have to transform the source letters before they reach the receiver. If the channel were perfect there would be a one-one correspondence between the letters of the two alphabets (and the associated probabilities) and our one-dimensional measure of information would suffice. However, if the channel is "noisy," i.e., if there is a finite probability that a letter from the source may be distorted into an erroneous receiver letter, then a more elaborate statistical model is required. We will assume that the effect of the noise may be described completely by specifying for each source letter the probabilities with which it is transmitted to the receiver. In a binary channel we might have a 0 appearing as a 1 with a probability 1-p, and a 1 appearing as a 0 with probability of 1-q

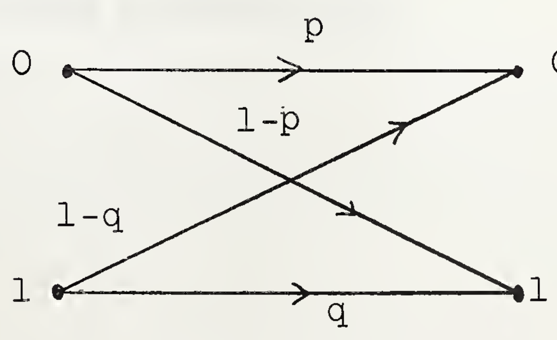


Fig. 3.4.2

A more general "black-box" model is shown below.

$$\begin{tikzcd}
\boxed{\text{SOURCE}} \arrow[r] & \boxed{\text{ENCODER}} \arrow[r] & \boxed{\text{CHANNEL}} \arrow[r] & \boxed{\text{DECODER}} \arrow[r] & \boxed{\text{RECEIVER}} \\
 & & \boxed{\text{NOISE}} \arrow[u] & &
\end{tikzcd}$$

Fig. 3.4.3

The encoder and decoder are included to make explicit the fact that the source and receiver may use different alphabets. As we shall see later, the encoder and decoder serve a much more important function in that a transformation of information may be desirable even within a <u>common</u> alphabet to reduce the effects of noise in the channel.

To sum up, then, the communication system works in the following way. The <u>source</u> selects letters (events) one at a time from its finite alphabet and passes them to the encoder which performs a transformation into the channel alphabet. The channel transmits the new letter to the decoder which transforms this into the receiver alphabet. The channel noise may distort the transmitted letter into any other letter of its own alphabet (with known probability).

## Section 3.5 Mathematical Model of a Communication System

We can formalise this description in the product space of two finite discrete schemes. Let X, Y be two abstract sets each containing a finite number of elements x and y respectively. Let p(x), p(y) be probability distributions defined over X and Y so that p(X) and p(Y) = 1. We have as before

$$H(X) = -\sum_{X} p(x)\log p(x)$$

and $$H(Y) = -\sum_{Y} p(y)\log p(y)$$

the entropies (average amounts of information) at the source and receiver respectively.

Let $X \otimes Y$ be the finite space consisting of all pairs (x,y) and let p(x,y) be a probability distribution over $X \otimes Y$. Then we have probability distributions $p(x) = \sum_{Y} p(x,y)$ over X

and $p(y) = \sum_{X} p(x,y)$ over Y

and we can write

$$H(X,Y) = -\sum_{X}\sum_{Y} p(x,y)\log p(x,y)$$

The conditional probability $p(x|y) = \frac{p(x,y)}{p(y)}$ for $p(y) > 0$ gives rise to a probability distribution over X and hence to a conditional entropy

$$H(X|Y) = \sum_{Y} p(y)H(X|y)$$

$$= -\sum_{Y}\sum_{X} p(x,y) \log p(x|y).$$

For completeness we note that

$$H(Y|X) = -\sum_{X}\sum_{Y} p(x,y)\log p(y|x).$$

Many relationships exist between these entropies. In particular,

$$H(X,Y) = H(X|Y)+H(Y)$$

$$H(X,Y) = H(Y|X)+H(X)$$

and $H(X) \geq H(X|Y)$

The first two may be shown by substitution of $p(x,y) = p(x|y)p(y) = p(y|x)p(x)$ in the definition of $H(X,Y)$ and using

$$p(x) = \sum_{Y} p(x,y) \text{ and } p(y) = \sum_{X} p(x,y)$$

The inequality, (due to Shannon), may be demonstrated as follows:

$$H(X|Y) - H(X) = -\sum_X\sum_Y p(x,y) \log (p(x|y))$$

$$+\sum_X p(x) \log p(x)$$

$$= -\sum_X\sum_Y \{p(x,y)\log p(x|y) \ -p(x,y)\log p(x)\}$$

$$= \sum_X \sum_Y p(x,y)\log \frac{p(x)}{p(x|y)}$$

$$\leq \sum_X\sum_Y p(x,y)\{\frac{p(x)}{p(x|y)} - 1\}\log e$$

since $\log x \leq (x-1)\log e,\ x > 0$

$$\leq \sum_X\sum_Y \{p(x)p(y)-p(x,y)\}\log e$$

$$\leq \sum_Y \{p(y)-p(y)\}\log e$$

$$\leq 0.$$

The function

$$H(X)-H(X|Y)(= \sum_X\sum_Y p(x,y)\log \frac{p(x|y)}{p(x)})$$

is important as it provides a measure of the information transmitted through the channel. Its form suggests the following definition. The mutual information conveyed by the pair (x,y) is

$$I(x;y) = \log \frac{p(x|y)}{p(x)} \left(= \log \frac{p(x,y)}{p(x)p(y)}\right).$$

I(x;y) is a measure of the information provided by the element y about the element x, in the sense that, from the viewpoint of the receiver, knowing the probability that x is sent and knowing the conditional probability that y is received if x is sent then the logarithm of the ratio of these probabilities indicates the gain in information at the receiver.

Note that as one would expect $I(x;y) = I(y;x)$; this justifies the definition of mutual information given above. Also $I(x;x) = -\log p(x)$ which conforms with the original I function we used earlier to develop the entropy function.

The average of the mutual information of all the pairs (x,y) per transmitted letter is then

$$I(X;Y) = \sum_X \sum_Y p(x,y)\ I(x;y)$$

$$= \sum_X \sum_Y p(x,y) \log \frac{p(x|y)}{p(x)} .$$

As we have seen $I(X;Y)(=H(X)-H(X|Y))$ is essentially non-negative although the individual $I(x;y)$'s may be negative.

## Section 3.6 Channel Capacity, Redundancy

One of the central concepts of information theory is channel capacity which was introduced by Shannon [1]. Channel capacity is the maximum rate of transmission of information through a channel in bits per letter and is defined mathematically as follows.

$$C = \max I(X;Y)$$

$$= \max \{H(X)-H(X|Y)\} .$$

Where the maximisation is with respect to all possible sets of probabilities associated with the source.

For example, in the discrete noiseless channel $I(X;Y) = H(X)$, hence $C = \max H(X)$

$$= \max \{-\sum_X p(x)\log p(x)\}$$

but H(p) takes on its largest value when all probabilities are equal. Hence if there are n letters in the source (receiver) alphabet then C = log n bits per letter.

That a maximum I(X;Y) does exist can be shown if we consider C as a continuous function of the n variables $p(x_1)$, $p(x_2), \ldots, p(x_n)$ in the general case. The $p(x_i)$ must satisfy the conditions $p(x_i) \geq 0$ and

$$\sum_{i=1}^{n} p(x_i) = 1 \text{ which determine a}$$

closed bounded set of points in n-dimensional Euclidean space. Hence C possesses a maximum and a minimum value for some sets of p(x).

The difference between the channel capacity (maximum rate) and the actual rate I(X;Y) is defined as the <u>redundancy</u> of the communication system.

$$\text{Redundancy} = C - I(X;Y).$$

The <u>relative redundancy</u> is defined as

$$\text{Relative Redundancy} = \frac{C - I(X;Y)}{C}$$

The <u>efficiency</u> of a channel is defined as

1-relative redundancy.

In the general case, the computation of channel capacity

$$C = \max\left(\sum_X \sum_Y p(x,y)\log \frac{p(x|y)}{p(x)}\right)$$

involves a long calculation using Lagrange multipliers but is not particularly difficult.

## Section 3.7 Binary Symmetric Channel

The channel which is of most interest to us in connection with digital computers and also because it is a very simple channel, is the binary symmetric channel (B.S.C.). It is illustrated below:

Source letter {0,1}    Receiver letters {0,1}

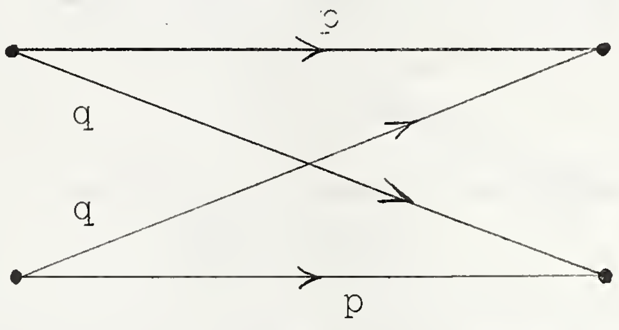


Fig. 3.7.1

At the source, $p(0) = \alpha \quad p(1) = 1-\alpha$

and $\quad p(0|0) = p(1|1) = p$

$\quad p(0|1) = p(1|0) = q(=1-p)$

Then $H(X) = H(\alpha,1-\alpha) = -\alpha\log\alpha-(1-\alpha)\log(1-\alpha)$

$$H(X|Y) = \sum_{j=1}^{2}\sum_{i=1}^{2} p(x_i)p(y_j|x_i)\log p(x_i|y_j)$$

$$=-\{p(0)p(0|0)\log p(0|0)+p(1)p(0|1)\log p(1|0)$$

$$+p(0)p(1|0)\log p(0|1)+p(1)p(1|1)\log p(1|1)\}$$

$$= -\{\alpha p\log p+(1-\alpha)q \log q + \alpha g \log q+(1-\alpha)p\log p\}$$

$$= - p \log p-q \log q \quad (\text{independent of } \alpha)$$

$$C = \max_{\alpha,1-\alpha}\{H(\alpha,1-\alpha) + p \log p+q \log q\}$$

$$= 1 + p \log p + q \log q.$$

The capacity of the general binary channel

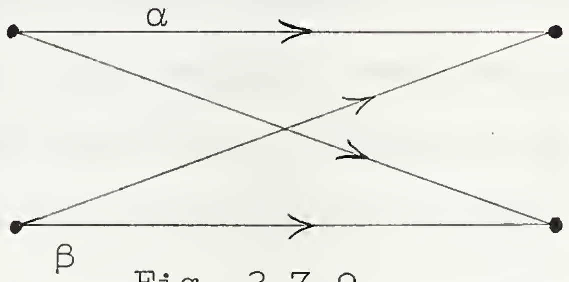


Fig. 3.7.2

has been given by Silverman and Chang as

$$C(\alpha,\beta) = \frac{-\beta H(\alpha,1-\alpha) + \alpha H(\beta,1-\beta)}{\beta-\alpha}$$

$$+ \log \{1 + \exp \frac{H(\alpha,1-\alpha)-H(\beta,1-\beta)}{\beta-\alpha}\}$$

where the source probability p(0) leading to the channel capacity is given by

$$p(0) = \beta(\beta-\alpha)^{-1}-(\beta-\alpha)^{-1}\left[1+\exp \frac{H(\beta,1-\beta)-H(\alpha,1-\alpha)}{\beta-\alpha}\right]^{-1}$$

and $p(1) = 1-p(0)$. Notice that to achieve channel capacity we must use the source probabilities defined by the channel noise characteristics. This point is not too clear in the symmetric case.

Binary channels have been most extensively studied.

## Section 3.8 Nth-order Extension of Channel

Channels such as we have described are seldom used to transmit single letters but rather sequences of letters one after another representing some message or other. That is, we are interested in the "Nth order extensions" of the basic channels.

If we transmit n successive letters x through a channel we can consider this as a new channel whose "letters" consist of the set U of all sequences u of length n of x's and whose set V of received letters consists of all sequences v of length n of y's. The probability distribution $p(v|u)= p(y_1|x_1)p(y_2|x_2)\dots$ $.p(y_n|x_n)$ and $u=(x_1,x_2,\dots,x_n)$, $v=(y_1,y_2,\dots,y_n)$ on

the assumption that successive letters are independent. Such a channel is described as "memoryless." (Note that the $x_i, y_i$ may take on any of the values of the X and Y spaces respectively.)

Let us take as a source probability distribution $p(u)=p_1(x_1)p_2(x_2),\ldots,(p_n(x_n)$ where the $p_i(x_i)$ are distributions over X.

We have after some algebra

$$\log p(u) = \sum_{i=1}^{n} \log p_i(x_i)$$

$$\log p(u|v) = \sum_{i=1}^{n} \log p_i(x_i|y_i)$$

$$H(U) = \sum_{i=1}^{n} H_i(X)$$

$$H(U|V) = \sum_{i=1}^{n} H_i(X|Y).$$

From this it can be shown that the capacity of the new channel $C_n$ is

$$C_n = \max I(U;V) = \max\{H(U)-H(U|V)\}$$

$$= n \max I(X;Y)$$

$$= nC$$

where C is the capacity of the original channel.

Feinstein and Fano [2] showed this result to be true for a general probability distribution p(u) not just the simple product distribution which we assumed.

As we have seen the more general channel has not introduced anything new except perhaps a large increase in computational labor in analysis. However we can now treat much larger alphabets with the same mathematical tools. For instance, instead of two letters in the binary channel we have available $2^N$ letters (or messages) - a much more useful capability. With the extended channel it can be shown that it is possible to transmit information at a rate $<C$ in a noisy channel and guarantee that it will be received with an arbitrarily small probability of error. To do this it is essential to send long sequences of letters. Unfortunately, at the moment it is rarely possible to make use of long sequences to improve transmission within digital computers or in most current transmission systems because of prohibitive cost of equipment. Strangely enough, digital computers seem to be about to come to the rescue of information theory in this particular problem. Theycan serve as two of the important "black-boxes" of our communication channel, the encoder and decoder and will become economically justifiable in the near future.

Even if this particular problem were not solved, information theory would still have much to contribute in the form of encoding theory. The study of encoding is one of the most highly developed parts of information theory as the astonishing number of papers available will attest. Encoding is important not only in practical applications but also is essential to the proof of our assertion of being able to guarantee transmission through a noisy channel.

## Section 3.9 Concepts of Encoding and Decoding

Encoding is any procedure (perhaps random) for associating messages expressed in one alphabet in a one-one manner with a set of messages in another (or the same) alphabet. Encoding may be used for a number of purposes. One we examined in the last chapter was to make communication possible between the decimal language used by humans to the binary language used by computers. A more important function is to re-express information in a form more likely to resist the effect of noise in a channel. Even if we assume a noiseless channel, we can use encoding to convert information into its most efficient (e.g. compact) form.

Decoding is the inverse operation of recovering the original messages from the transmitted messages. Decoding in the presence of noise is a considerably more difficult

operation than encoding, as one would expect.

A schematic diagram (adapted from Fano) illustrates the concept of encoding and decoding.

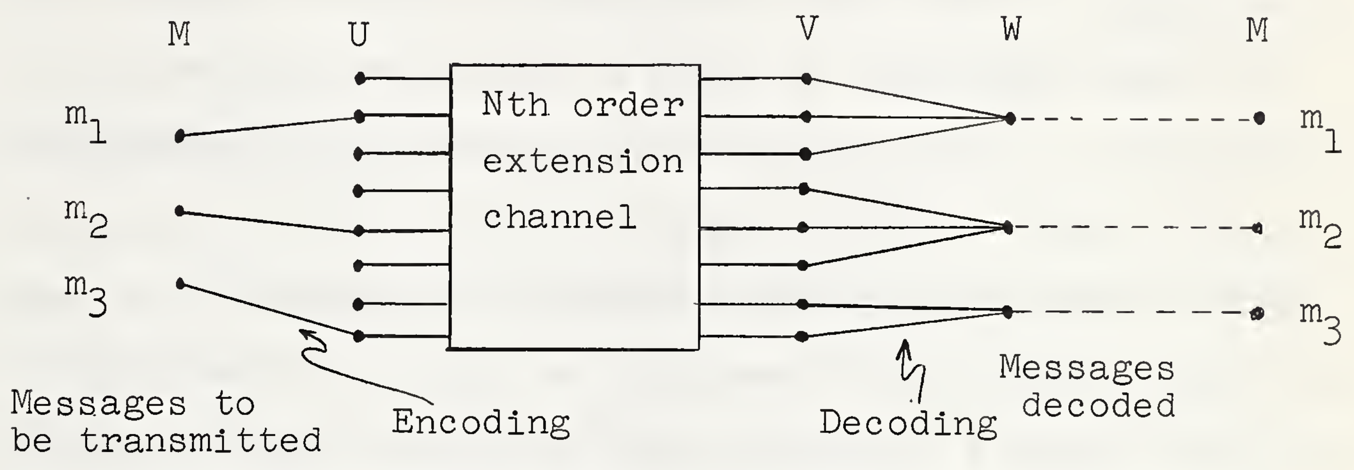


Fig. 3.9.1

Encoding is a one-one mapping from the message space M into the input space of the channel (U). An intelligent choice in encoding will obviously make the task of decoding easier (or even possible). Decoding is a many-one mapping from the channel output space (V) into the message space again, via the W space. The W space is included merely to indicate that the alphabet in which the messages finally appear need not be that of the message space M but are simply in one-one correspondence with them. The decoder is in effect a decision

scheme which associates sets of the output sequences V with the source messages as best it can.

To see how such a descision scheme might work, we can consider U as an n-dimensional space and u = $(u_1, u_2, u_3, \ldots, u_n)$ obviously represents a point in this space with the coordinates $u_i$ restricted to the values 0 or 1. All of the points lie on the n-cube and there are $2^n$ of them. Let us take as our measure of distance the minimum number of edges of the n-cube that must be traversed to go from one point to another. Hence two points which differ in r digits will be considered as a distance of r apart. Then if a vector u has r or less of its digits altered by noise then the point u may be "moved" anywhere within a distance of r by the noise of the point u. (The set of points which lie within a distance r of a given point may be conveniently termed a sphere of radius r but, as Golomb says, it is not everyone's intuitive notion of a sphere.) Hence if we choose a set of u vectors which are at least 2r + 1 distance units apart to represent the messages M then we can decode correctly received vectors with r or less errors in them.

Clearly, n,r and the maximum number of messages which can be decoded correctly are closely related. In fact N the number of disjoint spheres which can be packed into the n-cube is

given by

$$N = \text{Integer part of} \left\{ \frac{2^n}{\sum_{j=0}^{r} \binom{n}{j}} \right\}$$

(~~due to Hamming.~~)

Of course, this decoding scheme will not guarantee perfect decoding if there are more than r errors but this scheme is used as a basis for practical decoding procedures as we shall see in the next chapter. What we would like to be able to do is find an encoding scheme which will permit the messages to be transmitted with an arbitrarily small probability of error. No such scheme has been found but it is possible to find an upper bound for the probability of error by evaluating the average error over the ensemble of encodings generated by a random assignment of channel sequences to messages.

For each message $m_i$ a channel sequence is constructed by selecting n letters completely at random with equal probabilities. The reason for this random procedure is that the n digits sent through the channel and the n received digits for each message are statistically independent and equiprobable. Conceivably, the letters in a particular mapping might not be

independent but over the ensemble of all such mappings it would be true.

An extremely large number of such mappings are constructed (a thought experiment) and provided to both encoder and decoder to be used for successive messages. Each would know which mapping was in use for any particular message. It can be shown that the frequency of decoding errors will converge to the average probability of error evaluated over the entire ensemble of mappings, using a decoding criterion such as we outlined for the binary channel. The average probability of error remains bounded as $n \to \infty$ or, equivalently, the probability of correct decoding can be made as large as desired.

## Section 3.10 The Fundamental Theorem of Information Theory

The above remarks, or rather their mathematical equivalent, constitute a proof of the existence of an encoding scheme with a probability of error which can be made as small as desired. The existence of such a scheme is the first step in the proof of the fundamental theorem of information theory which may be stated as follows:

Theorem.

Let C be the capacity of a discrete finite constant memoryless channel and $R < C$, and S be a discrete independent

source with a specified entropy. It is possible to encode the output of S for transmission through the channel at the rate R and to decode the information transmitted with as small a probability of error as desired.

The proof of this theorem is perhaps the main theme of information theory. The theorem has been proved by many different methods since the original proof by Shannon [1] in 1948. It has been put on a sound mathematical basis by Feinstein, Wolfowitz and others and continues to occupy much attention as far as generalisation is concerned. The words "discrete," "finite," memoryless" suggest immediate generalisations which are in various states of accomplishment.

The proof of the theorem is not presented for the simple reason of its complexity. Fano gives a proof for the binary symmetric channel occupying 25 pages of his book "Transmission of Information." More general proofs are even longer.

## Section 3.11 Theoretical and Practical Implications of the Fundamental Theorem

The theoretical importance of the theorem can hardly be underestimated. It perhaps suffices to say that it represents the justification of a great deal of labor in

a particularly difficult field of mathematics. More than that, however, it states that encoding schemes exist by means of which perfect transmission may be achieved. The so-far unsuccessful search for such coding schemes might be less eagerly pursued without its assurance of their existence.

The converse of the theorem has also been proved (Wolfowitz) and asserts that transmission at rates higher than C is not possible. This acts as a useful check on theoretical results.

Despite its great interest theoretically, the value of the theorem is limited from an immediately practical point of view. First of all, no practical encoding procedure has been discovered and even if it were the theorem still requires large values of n to come close to achieving theoretical channel capacity. Fano [1] estimates that values of n of about 50 to 100 would be necessary to make binary channels transmit accurately at substantially greater rates than could be achieved by conventional engineering improvements. Using large values of n raises the question of how fast the complexity of the necessary electronic equipment rises with n. Wozencraft has devised a code for which the equipment complexity required for decoding increases only as nlogn . The complexity of current equipment is

more likely to grow exponentially with n (Peterson). As an indication of the present situation, a standard code of 8 binary digits per message was recently agreed on as being adequate for future use. (Bemer ) It appears that the difficulties of implementing long message-lengths are primarly engineering problems such as cost but in special areas such as communication with satellites at extreme distances or at critical points of a flight such as re-entry where extreme noise conditions will be encountered it is likely that elaborate encoding and decoding devices will be the only available solution (Dimsdale.)

The study of digital computers from an information theoretic point of view is not far advanced. We made the somewhat specious remark at the beginning of this chapter that computers could be regarded as a set of transmission channels. However, the channels are interconnected in a fairly complex way and tools for studying such a system as a whole are simply not available. If we applied our simple theory to individual channels in the system to make them more efficient it is possible that local increases in efficiency would have little effect on the over-all system. In addition, the noise environment in the digital computer is more favorable than, say, in long-distance communication and is still easily controlled by conventional engineering

techniques. It is possible that computers which have to operate for long periods in inaccessible places such as outer space will require the techniques of information theory to ensure reliability.

At the moment it would be fair to suggest that digital computers are more useful to information theory than vice versa. Computers are useful tools for carrying out "proofs by exhaustion" where theoretical methods are not yet available. They have also been used as ready-made general-purpose encoders and decoders.

## Appendix 3 Encoding for the Noiseless Channel

Although the assumption that a channel is noiseless is not a realistic one, the study of encoding for such channels is one of the more fascinating by-ways of information theory since it has interesting connections with other fields of mathematics. The codes developed for the noiseless channel throw some light on the coding problem for the noisy channel and, oddly enough, some of them have error-limiting properties which might make them useful in a noisy environment.

It was suggested in Section 3.1 that a code is most efficient if it minimises the function

$$\sum_{i=1}^{n} p(E_i)L(E_i)$$

which is actually the average number of digits in the sequences of a code. Under a certain restriction it is possible to show that this function has a lower bound. The restriction is that the code should be <u>uniquely decipherable</u>, i.e., it is possible to separate the code sequences from each other when they are transmitted without space-marks or special separator symbols. For example, the words "inform," "at," "ion" when transmitted without spacer-marks, form another word "information" which is not implied by the three separate words.

<u>Theorem</u> (McMillan)

Let $\{m_1, m_2, \ldots, m_N\}$ be a set of messages encoded in uniquely decipherable sequences of lengths $\{n_1, n_2, \ldots, n_N\}$ of letters taken from the D-letter alphabet $\{a_1, a_2, \ldots, a_D\}$. Then

$$\sum_{i=1}^{N} D^{-n_i} \leq 1$$

Conversely, if integers $n_i$ exist satisfying this inequality, then it can be shown that a uniquely decipherable code exists (Sardinas and Patterson.)

For uniquely decipherable codes, it can be shown that a minimum exists for the average length of encoded sequences.

<u>Theorem</u> (Reza p. 148)

Let $\{X\} = \{x_1 x_2, \ldots, x_N\}$ be a set of messages with associated probabilities $\{p(x_1), p(x_2) \ldots\}$. If the message $x_i$ is encoded into a sequence of length $n_i$ of letters selected from the finite alphabet $\{a_1, a_2, \ldots, a_D\}$ then the average length of encoded sequences

$$\overline{L} = \sum_{i=1}^{N} p(x_i) n_i > \frac{H(X)}{\log D}$$

It is therefore possible to define efficiency for such codes:

$$\text{Efficiency} = \frac{H(X)}{\log D} \div \bar{L} = \frac{H(X)}{\bar{L} \log D} .$$

Huffman [2] gives a constructive method of finding codes with maximum efficiency. No such encoding procedure is available for the noisy channel.

# CHAPTER 4

# Error-Detecting and Error-Correcting Codes.

## Section 4.1 Introduction:

In the last chapter we saw that the fundamental theorem of information theory held out the prospect of perfect transmission of information despite the presence of noise. Yet, today, 15 years after its first appearance, no practical encoding methods are available which will allow its promise to be realized. The theorem in effect guarantees to correct all errors if we use long enough sequences. Now, if we "back off" from this absolute guarantee of perfect error-correction and take note of the practical problem of handling extremely long sequences of digits with present engineering techniques, we can approach the problem with different methods and perhaps come upon the fundamental theorem in a new form. (Peterson, p. 82.)

The new methods we shall discuss in this chapter are deterministic, as opposed to the purely probabilistic methods of Chapter 3. Of course, it is not possible to ignore the fundamentally probabilistic nature of the transmission of information in evaluating the usefulness of the codes that result from deterministic methods. The methods are deterministic in the sense that finite pre-specified procedures

are used to correct a few of the more likely errors which may occur in the channel. The essential idea is that additional digits which are functions of the information digits are sent through a channel with the information digits.. These digits do not increase the information content of a message but ensure that chosen error patterns will be corrected. The added digits increase the probability that the message will be decoded correctly; hence, it is not possible to separate completely the deterministic and probabilistic aspects of the problem. Further, the probabilistic properties of the channel will have to be taken into account in devising decoding schemes.

For simplicity, we will again choose the binary symmetric channel as a model. The more realistic asymmetric channels are beginning to receive some attention in the literature but as yet very few results are available as compared with those for the binary symmetric case. Many of these results may be generalised to channels with a prime number of letters or characters. These generalisations follow because the methods presented depend only on the fact that the binary characters {0,1} form a finite field under certain definitions of addition and multiplication (See Appendix 4a)

## Section 4.2 Parity Check Codes

The codes which we shall study are the so-called parity-check codes. They originated in a paper by Hamming in 1950 which is still the best short introduction to the subject. The codes are used in the following way: the encoder receives k binary digits of information from a source, computes n-k checking digits and transmits the n digits to a binary symmetric channel. The decoder recomputes the check digits from the n digits it receives and compares then with the received check digits. If they differ, one or more errors have occurred and the decoder may signal that an error has been detected and (perhaps) request a retransmission or it may have enough information to correct the errors. We shall amplify this point later but the following example illustrates the main ideas. A schematic diagram for n=7, k=4 is given in Fig. 4.2.1.

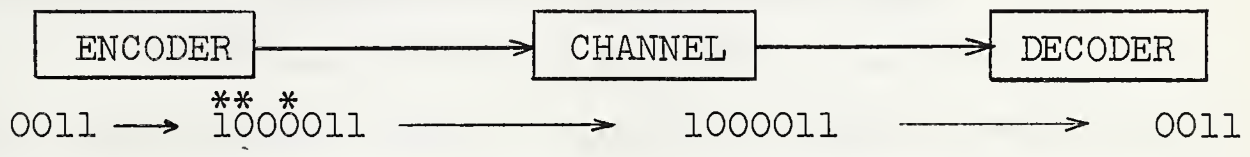


Fig. 4.2.1

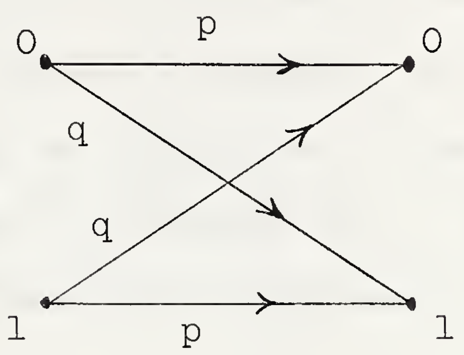
The checking digits inserted by the encoder and stripped off by the decoder are marked "*".

The presence of the n-k checking digits which contain no additional information reduce the effective channel capacity to at most k/n bits per binary digit. For useful values of k and n, this rate is substantially less than the maximum guaranteed by the fundamental theorem of information theory. However, no encoding schemes have yet been discovered which do approach the maximum rate. It has been pointed out that n must be quite large to guarantee the maximum but there are a number of special applications such as digital computers where small values of n are required. The parity-check methods work well for small values of n as well as reasonably large values, although usually at a cost in channel capacity. Until better methods are developed, the loss in channel capacity will have to be accepted as unavoidable.

## Section 4.3 Probability of Correct Decoding

Parity-check codes guarantee that a few of the more likely errors such as single or double errors will be corrected. Before examining the codes themselves let us consider whether this is a worthwhile capability. In an n-digit sequence, there are $\binom{n}{r}$ possible erroneous sequences that could result from r simultaneous crossover errors, i.e., 0 becomes 1 or 1 becomes 0. In a binary

symmetric channel the probability of r crossovers occurring is given by $q^r(1-q)^{n-r}$ where q is the probability of a crossover in each binary digit.



$(q \leq \frac{1}{2})$

Fig. 4.3.1

We have

$$\sum_{r=0}^{n} \binom{n}{r} q^r (1-q)^{n-r} = 1.$$

If no error correction is applied, the probability of receiving an incorrect n-sequence is

$$(1) \qquad \sum_{r=1}^{n} \binom{n}{r} q^r (1-q)^{n-r} = 1-(1-q)^n$$

$$= nq - \frac{n(n-1)}{2} q^2 + \ldots$$

$$\approx nq \text{ when } nq \ll 1.$$

If we apply a single-error correcting scheme, the probability of receiving an incorrect sequence is

(2) $$\sum_{r=2}^{n} \binom{n}{r} q^r (1-q)^{n-r} = \tfrac{1}{2} n(n-1) q^2 + \ldots$$

For values of n and q of practical interest, the probability of receiving an incorrect sequence is reduced satisfactorily. Strictly speaking, we should use k, the number of information digits in equation (1) instead of n. As we shall see, for a single-error-correcting code, k and n are related by

$$k \leq n - \log(1+n).$$

Taking this into consideration does not affect our conclusion significantly if we take k as large as is feasible. In fact, it can be shown for values of kq less than about 0.3, single-error-correcting codes give a lower probability of receiving and incorrect sequence than do "straight" codes (Reza, p. 175). For values of kq larger than 0.5 applying a single-error-correcting scheme makes it more likely that an incorrect sequence will be received.

With this assurance that for practical values of k, n and q, the application of error correction improves the chances of transmitting information through the B.S.C., let us consider how the codes work.

## Section 4.4 Definition and Uses of Parity Checks

A parity check is a digit appended to a sequence

of digits so that the new sequence contains an even or odd number of 1's. For example, the sequence 01011 contains three 1's so the even parity check for the sequence would be 1. Parity checks are normally placed to the right of the information digits so that the parity-checked sequence would be 010111. The reason for placing the checks to the right of the information digits is that, by convention, the order in which sequences are received or transmitted is in the written order, left to right, the leftmost digit being received or transmitted first. Hence, the information digits need not be stored in an encoder in order to calculate the parity check.

A single parity check would enable a decoder to tell if a single, triple, quintuple... error had occurred. Parity checks need not extend over the whole sequence and several parity digits which check subsets of the sequence of digits in a systematic manner permit error correction as well as error detection.

The choosing of parity checks in an efficient manner is a major problem of coding theory; efficient in the sense of providing the maximum feasible error-detecting and correcting capability with as few parity checks as possible.

To give substance to these ideas, let us examine in detail a parity-check code. The following code is a single-error correcting code with n=5, k=2 (number of information digits)

| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Table 4.4.1

$x_1$ and $x_2$ are information digits and $x_3, x_4, x_5$ are even parity checks.

$x_3$ checks $x_1$

$x_4$ checks $x_2$

$x_5$ checks $x_1$ and $x_2$.

To verify that the code may be used to correct single errors, we form the table of the code sequences listed in columns with the sequences resulting from a single error in each of the five positions.

| 0 0 0 0 0 | 0 1 0 1 1 | 1 0 1 0 1 | 1 1 1 1 0 |
|---|---|---|---|
| 0 0 0 0 1 | 0 1 0 1 0 | 1 0 1 0 0 | 1 1 1 1 1 |
| 0 0 0 1 0 | 0 1 0 0 1 | 1 0 1 1 1 | 1 1 1 0 0 |
| 0 0 1 0 0 | 0 1 1 1 1 | 1 0 0 0 1 | 1 1 0 1 0 |
| 0 1 0 0 0 | 0 0 0 1 1 | 1 1 1 0 1 | 1 0 1 1 0 |
| 1 0 0 0 0 | 1 1 0 1 1 | 0 0 1 0 1 | 0 1 1 1 0 |

Table 4.4.2

By inspection, no sequence appears twice in the table, hence a single error in any code sequence results in a unique sequence which can be "traced back" to the original sequence. To see how to trace it back without storing all of these sequences in a decoder, we revert to the parity checks. The decoder recalculates the parity of $x_3, x_1$; $x_4, x_2$ and $x_5, x_2, x_1$ in sequence. If the transmission is correct, then the parity of each set will be even. Let us denote even parity by 0 and odd by 1. Then the sequence resulting from checking a correct transmission would be 000. If, however, there was an error e.g., if, say, 10001 was received instead of 10101, the sequence generated by checking 10001 would be 100. It can be shown that a unique 3-digit sequence is generated by each of the five possible single errors independently of which code-sequence was sent. This sequence is variously called a "syndrome," a "parity-check vector" and "corrector." The decoder then has enough information to locate and correct the error on the assumption that a single error has occurred. In general, for a code with k information digits and m(=n-k) checking digits, the corrector must describe m+k+1(=n+1) different things, viz. the positions of the errors or an indication that no error has occurred. Since there are $2^k$ possible values of the corrector that

can be generated by k parity checks, then

$$2^m \geq m+k+1$$

i.e., $$2^{n-k} \geq n+1$$

$$\therefore\ 2^k \leq \frac{2^n}{n+1}$$

which is the condition on k, the number of information digits in a single error correcting code, referred to earlier.

These codes are often referred to as (n,k) codes and our example would be called a (5,2) code.

## Section 4.5 Geometry of Parity Checks

The effect of parity checks on messages can be illustrated geometrically; the geometric approach is simple and perspicuous and suggest new lines of enquiry which intersect enquiries in established disciplines. Suppose we wished to transmit the messages "yes" or "no" through a binary channel. If there is no noise in the channel we could agree to transmit "0" for "yes"and "1" for "no" and the coding problem is solved. Trivially, we can place the two messages at 0 and 1 of a one-dimensional space

Fig. 4.5.1

If the channel is noisy, then let us add an even parity check to each message so that 0 becomes 00 and 1 becomes 11. Geometrically in 2 dimensions this could be illustrated as:

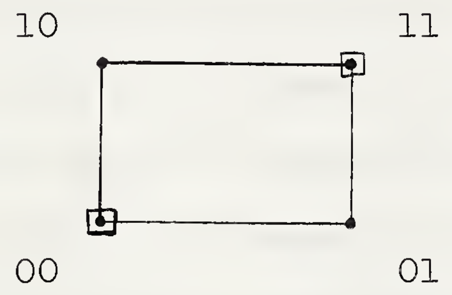
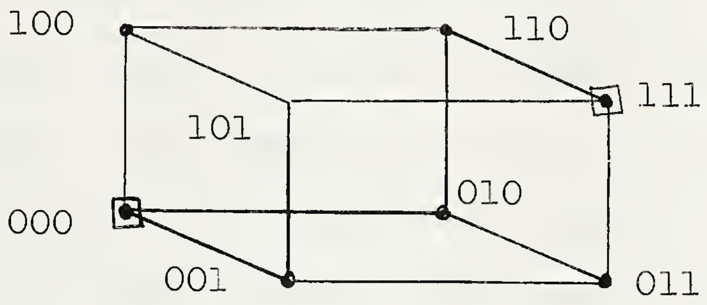


Fig. 4.5.2

Adding another parity check which is independent of the previous one (i.e., it checks the original message digit again), we have 000 and 111 as our encoded messages. In 3 dimensions we have:



Clearly the parity checks are pushing the code points further apart. The value of this lies in the fact that a single error in the message 000 would "move" the code-point to 100, 010 or 001 which are still closer to 000 than to 111. Hence, if we assume that single errors are most likely to occur, we can associate the set 000, 001,010, with 000 and 111,110,011,101 with 111 - a single error

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

correcting scheme of decoding. Of course, a double error would be decoded incorrectly. We could also use this as a single-and double-error detecting scheme if the decoder makes no attempt to correct errors.

The geometrical picture and the simple decoding scheme quickly get out of hand as the number of digits is increased and we have to seek more subtle methods. However, the important concept of Hamming distance is suggested by the simple ideas we have developed. Briefly, the distance between two points is the number of digits in which the two points differ e.g., 101 and 000 are a distance of 2 apart. Geometrically, the Hamming distance is the smallest number of edges of the n-cube that we must traverse to go from one point to the other. This concept of distance is more useful than the usual Euclidean concept in connection with codes. The Hamming distance is used in analysing the error detecting and correcting properties of codes. It is easy to see that choosing code-points at a distance of 1 apart does not permit any error detection, as in Fig. 1. In Fig. 3, we chose 000 and 111 which are 3 apart as our code points and noted that we could correct any single error. There are four pairs of points on the 3-cube which have this property. We could also choose four of the eight points and applied a single error detecting scheme, if the points are

at least a distance of 2 apart. In general, it can be shown (Hamming) that choosing codepoints a minimum distance of $2j+1$ units apart will allow the correction of $j$ errors and that $j$ errors may be detected if code points are chosen $2j$ apart.

At this point, an important question suggests itself; given $n$ and $j$, what is the largest number of codepoints in a $j$-error correcting code? Using the result on p. 61 of Chapter 3, Hamming showed that the number of disjoint shperes of radius $j$ which can be packed into the n-cube is an upper bound on $B(n,2j+1)$, the maximum number of codepoints in a j-error-correcting code. For an $(n,k)$ code, $B(n,2j+1) = 2^k$ where $k$ is the number of information digits. Hence

$$B(n,2j+1) = 2^k \leq \frac{2^n}{\binom{n}{0}+\binom{n}{1}+\dots+\binom{n}{j}}$$

More precise bounds have been found and are discussed in Appendix 4.b, but no general methods for constructing codes which attain these bounds are available. Codes which do attain the bounds are called optimal or "close-packed" codes - the geometric approch has translated the coding problem into a "packing" problem. It is well to note here the difference between optimal and "optimum" codes. An optimum code is an $(n,k)$ code for which the probability of decoding is at least as great as for any other $(n,k)$ code with the same $n$ and $k$. A number of optimum codes have been found by exhaustive calculations on digital computers, for values of $n$ up to 15 (Fontaine and Peterson.) The search is complicated by the fact that it has not yet been proved

that a code which is optimum for one value of q (the channel probability of error) is also optimum for all others (Peterson p. 72.)

While these elementary arguments and intuition have led to a number of significant results, a more general viewpoint is required for further development.

## Section 4.6 Mathematical Definitions of Parity Checks

Mathematically, parity checks are the sums modulo 2 of selected digits of a binary sequence and are based on the table

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

Table 4.6.1

where the symbol $\oplus$ denotes modulo 2 addition. $\oplus$ is also the Boolean "exclusive-or" operator and is easily implemented by electronic devices which partly accounts for its choice in this context. It is easy to show that the elements $\{0,1\}$ form a commutative group under the operator $\oplus$. (See Appendix 4.a). Further $\{0,1\}$ form a field under $\oplus$ for addition and multiplication defined by the table:

$$0.0 = 0$$
$$0.1 = 0 \qquad \text{Table 4.6.2}$$
$$1.0 = 0$$
$$1.1 = 1 \quad .$$

The parity checks for the (5,2) example could be written as the set of (linear) equations,

$$x_1 \oplus x_3 = 0$$
$$x_2 \oplus x_4 = 0$$
$$x_1 \oplus x_2 \oplus x_5 = 0$$

Clearly the n-sequences we have discussed are ordered sets of field elements and with suitable definitions of addition and multiplication of n-sequences it can be shown that the set of all n-sequences over the field {0,1} form a vector space. These remarks suggest that coding theory and modern algebra are closely related and that coding theorists may look to the older discipline for methods to deal with coding problems.

This is indeed what has happened. Recent developments owe much to modern algebra as we shall see. A minor instance of the influence of modern algebra is the use of "point," "vector" and "sequence" to indicate the same concept.

## Section 4.7 Codes, Vector Spaces and Matrices

A code may be described as a set of vectors selected from the $2^n$ vectors which comprise the space of all n-vectors. The selection may be random but usually is made according to some rule which makes encoding and decoding as simple as possible. The simplest useful case is that of linear codes, i.e., the code is a subspace of the vector space of all n-vectors over the field $\{0,1\}$. The error correcting properties of linear codes are more easily analysed than those of non-linear codes for the following reason: the minimum (Hamming) distance between code vectors is easily found by inspection of the code vectors themselves. Without this property, the distances between every pair of vectors would have to be calculated to find the minimum. Since a linear code is a subspace, its vectors form a group under $\oplus$, i.e., the difference between any two vectors is another vector of the subspace. Hence the minimum distance between vectors in the group must be the least number of 1's in one of the non-zero vectors. To exemplify, the four vectors of our previous example

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Table 4.4.1 (bis)

form a vector space and therefore a linear code. The smallest number of 1's in any non-zero vector is 3. Hence the code may be used to correct single errors.

Clearly, the use of parity checks is simply a way of selecting vectors from the $2^n$ possible vectors in order to form a subspace. Since an (n,k) linear code is a subspace there must exist k linearly independent vectors which span the space i.e., the other vectors may be generated by linear combinations of the basis vectors. For example, in the (5,2) code we have used, we can select the two vectors 01011 and 10101 as a basis and the other two 00000, 11110 may be generated by adding 01011 to itself and 01011 to 10101 respectively. The advantage of this description becomes clearer when k is large, say, 20. Then for a (30,20) code there would be $2^{20}$ code vectors. These could be specified by a 30 x 20 matrix - a significant saving in space.

The basis vectors may be arranged in the form of a matrix; e.g., for a (5,2) code the matrix may be

$$G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

and encoding may be described as premultiplication of this matrix by the vector of information digits, where the inner product of two vectors x and y are defined as follows

$$(x_1,\dots,x_n),(y_1,\dots,y_n) = x_1 \cdot y_1 \oplus x_2 \cdot y_2 \oplus \cdots \oplus x_n \cdot y_n$$

e.g., $$[1\ 1] \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} = [1\ 1\ 1\ 1\ 0\ ]$$

G is called the generator matrix of the code.

Another description of a linear code is available from its parity check equations. In the example, we had the equations

$$x_1 \oplus x_3 = 0$$
$$x_2 \oplus x_4 = 0$$
$$x_1 \oplus x_2 \oplus x_5 = 0.$$

These may be written in matrix form

$$Hx = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = 0$$

H is called the parity-check matrix of the code. The two matrices are of course closely related. Explicitly,

$$GH^T = HG^T = 0 \qquad \text{---- (1)}$$

since every code vector generated by the rows of G must

satisfy the parity check equations. In general, for an (n,k) code, G, the generator matrix, is a (k x n) matrix and H, the parity-ckeck matrix, is an (n-k)xn matrix for which Eqn. (1) holds. We can now state the following theorem. (Peterson, p.32).

Theorem:

Let X be a linear code defined by a parity check matrix H. Then for each code vector of weight d (number of 1's in the vector), there is a linear dependence relation among the columns of H; and, conversely, for each linear dependence relation involving columns of H, there is a code vector of weight d.

The proof follows from the fact that if $x = (x_1, x_2, \ldots, x_n)$ is a code vector of X then

$$xH^T = 0$$

i.e., if the ith column of H is denoted by $h_i$, then

$$\sum_{i=1}^{n} x_i h_i = 0.$$

This is a linear dependence relation between all or some of the columns of H. The number of columns involved in this dependence relation will be exactly the number of

$x_i$'s which have the value 1, i.e., the weight of the code vector x. Hence a linear dependence relation must exist for each code vector and must involve d columns where d is the weight of the vector. Conversely, every linear dependence relation between the columns of H defines a vector x for which

$$\sum_{i=1}^{n} x_i h_i = 0$$

and hence $xH^T = 0$, the necessary condition that x is a code vector.

If we assume that no column of H is zero, each linear dependence relation corresponding to a code vector of weight d implies that each set of d-1 or fewer of the columns involved is a linearly independent set. The important corollary given below follows immediately.

Corollary:

A code defined by a parity check matrix H has a minimum distance at least d if and only if every combination of d-1 or fewer columns of H are linearly independent.

This corollary suggests a basis for a constructive method of finding parity check matrices for codes with a

specified minimum distance. However, the procedures which have been developed from this by Mcluskey and others are still too time-consuming for practical use. Mcluskey notes that the problem is equivalent to a linear-programming problem but the number of inequations which are involved is of the order of $2^n$ which become impracticable to solve for even small values of n.

## Section 4.8 Decoding of Linear Codes

Up to this point in the chapter we have been able to proceed along purely deterministic lines. However, we must now recall the probabilistic nature of the binary symmetric channel in order to decode intelligently. Essentially, we ask the question, which errors are most likely to occur in a code-vector? After answering this question, we can devise a deterministic decoding method which will correct the most likely errors.

The decoding of linear codes may be carried out by calculating the syndrome of a received vector v and associating it with a particular error. In terms of the parity check matrix the syndrome is the vector $s = vH^T$. s will have as many elements as there are parity check equations. The equations need not be independent but the

rank of H will determine the number of unique syndromes which can be calculated - this number is $2^r$ where r is the rank of H. Further, the syndrome is independent of the actual vector which the sender intended to be received. The effect of noise on a transmitted vector u may be described by adding a vector which has 1's in the positions of the errors in the received vector v, e.g.,

$$\underset{\text{Transmitted vector}}{(0\ 1\ 0\ 1\ 1)} \oplus \underset{\text{error vector}}{(0\ 1\ 0\ 0\ 0)} = \underset{\text{Received vector.}}{(0\ 0\ 0\ 1\ 1)}$$

Hence $v = u \oplus e$ where e is an error vector and $s = vH^T = (u \oplus e)H^T = eH^T$ since $uH^T = 0$. For an (n,k) code, the rank of H is m = n-k and there are $2^m$ possible errors which can be detected by use of the syndrome. The syndrome thus divides the $2^n$ error vectors into $2^m$ classes each of which contains $2^k$ vectors, each class having a characteristic syndrome. There is not enough information to select the particular error in each error class so a principal error in each class must be chosen - the choice of the principal errors will depend on the purpose of the code. The most common choice is to select error vectors such that the probability of correct decoding is largest assuming that each code vector is equally likely to be

transmitted. For example, if the error vectors 00100 and 11010 were associated with the same syndrome, it would be preferable to choose 00100 as the principal error since one error is more likely to occur than three simultaneous errors.

We again turn to modern algebra for a method to handle this problem. This time the relevant concept is that of cosets or equivalence classes. Since an (n,k) code is a subgroup of the group of all n-sequences under the operation $\oplus$, it is possible to divide the $2^n$ vectors into $2^{n-k}$ distinct classes each containing $2^k$ vectors. These classes are called cosets and have the following property; if one of the vectors of a coset is associated with one of the code vectors, the other members of the coset can be found by adding the selected coset vector to each code vector in turn. In this way, each member of each coset is uniquely associated with a code vector in an easily calculated way. The selected coset vector is called the coset "leader." The cosets are the error classes referred to earlier, and the coset leaders are the principal errors of the error classes. The other members of a coset are the vectors which result when the principal error occurs in the transmission of code vectors.

Each member of a coset has the same syndrome as the coset leader, since

$c = \ell \oplus x$ where c = member of the coset corresponding to the code vector x and $\ell$ = coset leader.

and $cH^T = (\ell \oplus x)H^T$

$= \ell H^T$ = syndrome of $\ell$.

By way of illustration, the construction and use of an array of cosets is explained for the case of a(5,2) code.

The $2^n$ vectors are arranged in a standard form as an array of cosets. The procedure is easily explained by an example. For the (5,2) code, we arrange the code vectors (00000,01011,10101,11110) as column headings with the identity element (00000) in the leading position.

| | | | | Syndrome |
|---|---|---|---|---|
| 0 0 0 0 0 | 0 1 0 1 1 | 1 0 1 0 1 | 1 1 1 1 0 | (0 0 0) |
| 0 0 0 0 1 | 0 1 0 1 0 | 1 0 1 0 0 | 1 1 1 1 1 | (0 0 1) |
| 0 0 0 1 0 | 0 1 0 0 1 | 1 0 1 1 1 | 1 1 1 0 0 | (0 1 0) |
| 0 0 1 0 0 | 0 1 1 1 1 | 1 0 0 0 1 | 1 1 0 1 0 | (1 0 0) |
| 0 1 0 0 0 | 0 0 0 1 1 | 1 1 1 0 1 | 1 0 1 1 0 | (0 1 1) |
| 1 0 0 0 0 | 1 1 0 1 1 | 0 0 1 0 1 | 0 1 1 1 0 | (1 0 1) |
| 1 0 0 1 0 | 1 1 0 0 1 | 0 0 1 1 1 | 0 1 1 0 0 | (1 1 1) |
| 1 1 0 0 0 | 1 0 0 1 1 | 0 1 1 0 1 | 0 0 1 1 0 | (1 1 0) |

Table 4.8.1

For the leader of the second row we choose an element of

lowest weight in the vectors which have not yet appeared and add this to each of the code vectors in turn - the resulting vector is placed in the column below the code vector, e.g., (00001) $\oplus$ (01011) = (01010). The whole array is developed by choosing lowest weight vectors as row leaders from remaining vectors. No vector appears more than once since we started with a group (a linear code) and the rows are cosets. Hence if we have a table of syndromes and coset leaders the decoding can be considerably simplified. This particular array is constructed so that the vectors most likely to be received in a binary symmetric channel appear in the column corresponding to the transmitted vector.

It can be shown that this standard array leads to a maximum probability of receiving a correct vector when the code vectors are equally likely to be transmitted (Peterson p.37).

The decoding procedure then is to calculate the syndrome $vH^T$ from the n-digit received vector and look up the coset leader in the syndrome-coset leader table. This coset leader is the presumed error pattern and is added to the received vector. This gives the vector which was sent, provided a principal error occurs. For example, if

the vector 11010 were received,

$$S = vH^T = [11010] \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [100] .$$

The coset leader corresponding to (100) is (00100).
Then transmitted vector = (00100) ⊕ (11010)
= (11110).

In general, a decoder for an (n,k) code will be able to correct $2^{n-k}-1$ errors. The (5,2) code corrects 7 errors which are chosen as 7 of the most likely errors, viz., the 5 single errors and 2 of the possible 10 double errors.

The probability of receiving an incorrect sequence is

$$1 - (\underbrace{(1-q)^5}_{\text{No error,}} + \underbrace{5q(1-q)^4}_{\text{5 single errors,}} + \underbrace{2q^2(1-q)^3}_{\text{2 double errors,}})$$

where q is the channel probability of error.

Since the decoding scheme is based on the standard array, this is the minimum probability of error that can be achieved with any (5,2) code. Many of these "best"

(n,k) codes have been discovered and a large selection is listed in Slepian [1]; some of them appear in Appendix 4.c.

Many other codes based on these ideas have been developed as special cases and as generalisations. As an example of the latter, we might mention iterated codes in which two or more linear codes are combined to form a more powerful code. The structure of these codes may be suggested by the diagram.

| | |
|---|---|
| Information Symbols | Checks on rows |
| Checks on Columns | Checks on Checks |

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |

Fig. 4.8.1

The code on the right is used in IBM magnetic tape units for error correction and detection - it has a minimum weight 4. Kautz has devised a number of multiple error-correcting codes based on multi-dimensional arrays.

## Section 4.9 Cyclic Codes

One of the most interesting recent developments

is that of cyclic codes - a subclass of linear codes. These codes can be implemented with remarkably simple equipment and they should remove one of the major barriers to the widespread use of codes, i.e., the complexity of coding and decoding equipment. The word "cyclic" is used for certain of the unit-distance binary codes such as the Gray codes described in Chapter 2. The codes for this section are parity-check codes whose name derives from certain properties of the codes, one of which is that if $(x_1,x_2,\ldots,x_n)$ is a code vector then $(x_n,x_1,x_2,\ldots,x_{n-1})$ is also a code vector. This already simplifies encoding somewhat but the cyclic structure goes much deeper than this.

The connection between the general linear codes and the cyclic codes can be demonstrated if we consider the syndromes of an $(n,k)$ code with parity check matrix H. We saw that there were $2^m$ different syndromes, m of which were linearly independent and n of which could be the syndromes corresponding to single errors. Let us call these "single-error" syndromes $s_1,s_2,\ldots,s_n$, where $s_{n-i+1}$ is associated with an error in position $x_i$ (for notational convenience later). Since the $s_i$ are actually the columns of the m x n parity check matrix H, the condition that a vector x be a code vector, $xH^T = 0$, may be written as a vector equation

$$x_n s_1 \oplus x_{n-1} s_2 \oplus \ldots \oplus x_1 s_n = 0$$

Now, since there are m linearly independent $s_i$'s, it is possible to find an m x m matrix T such that $s_i = T^{i-1}s_1$ where $s_1 = (1,0,0,\ldots,0)$, i.e., pre-multiplication of $s_1$ by powers of T will generate the n "single-error" syndromes, $s_2, s_3, \ldots, s_n$, and $s_1 (= T^0 s_1)$. Hence the condition that a vector x be a code vector may be written

$$\sum_{i=1}^{n} x_{n-i+1} T^{i-1} s_1 = 0 \qquad \ldots.4.9.1$$

This may be used as a definition of a linear code (Abramson, [2]) and clearly the properties of the matrix T define the properties of the code, given m linearly independent $s_i$. This definition is useful in this context because it leads directly to the definition of cyclic codes.

A cyclic code is defined as a code whose matrix T is cyclic, i.e., $T^n = I$. It is easily shown that the cyclic property of the code vectors noted earlier follows from the cyclic nature of T. If we pre-multiply Equation 4.9.1 by T we have, for a code-vector $x = (x_1, x_2, \ldots, x_n)$,

$$T(x_n s_1 \oplus x_{n-1} T s_1 \oplus \ldots \oplus x_1 T^{n-1} s_1) = 0$$

or $$x_n T s_1 \oplus x_{n-1} T^2 s_1 \oplus \ldots \oplus x_1 T^n s_1 = 0$$

i.e., $$x_1 s_1 \oplus x_n T s_1 \oplus \ldots \oplus x_2 T^{n-1} s_1 = 0$$

which we shall adopt as the condition that $(x_n, x_1 \ldots, x_{n-1})$ be a code vector.

It has been found that codes generated by a general cyclic matrix T are sometimes difficult to implement. Meggitt showed that the codes could be transformed into a simpler form and retain their error-correcting properties. The appropriate transformations of the matrix T are similarity transformations $U = STS^{-1}$ under which the characteristic polynomial of T is invariant. U and T produce codes with identical properties if the minimum polynomials of U and T are identical with their characteristic polynomials. The matrix T can then be replaced by its companion matrix which happens to be remakably simple to implement in terms of certain circuits called linear sequential networks. The search for good codes can thus be narrowed to a search for suitable companion matrices or equivalently suitable characteristic polynomials.

The characteristic equation which an m x m matrix T satisfies may be written

$$T^m + c_{m-1}T^{m-1} + \ldots + c_1 T + c_o = 0.$$

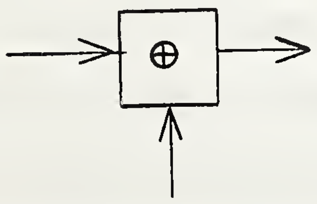
The corresponding companion matrix is simply

$$\begin{bmatrix} c_{m-1} & c_{m-2} & \cdots & c_1 & c_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 \cdots & 0 & 0 \\ \vdots & \vdots & & & \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} .$$

We will assume that the T matrices have this form in what follows. In the next section, a hardware realization of cyclic codes is presented.

## Section 4.10 Linear Sequential Circuits

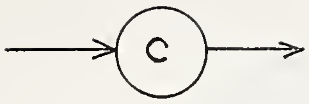
Linear sequential circuits are made up of elements which have been designed to carry out the basic functions we have been discussing viz., " $\oplus$ ", "." and a means of storing the value of a variable which may be 0 or 1. Schematic diagrams are an easy way of visualising the operation of such circuits and we will use the following conventions



Modulo 2 adder: implements the table

| $\oplus$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

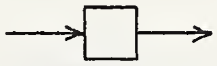
by providing the appropriate sum digit when two input variables are present on its input connectors.



Switch or multiplier: implements the table

| . | 0 | 1 |
|---|---|---|
| c = 0 | 0 | 0 |
| c = 1 | 0 | 1 |

by multiplying the incoming variable by a constant c. In the binary case, this is just the presence of a connection (c=1) or no connection (c=0). (If the values of the variables were the elements of a field of n elements, then the multiplication would appear explicitly.)



Storage element: holds the two states 0 or 1 indefinitely until changed by a new value appearing at the input arrow. The value the element has may be transmitted to another storage element, adder or multiplier without changing the value in the element.

Since the circuit elements in practice require a certain amount of time to reach their steady states after being activated, the transfer of values from one element to another, and the operations + and . are valid only during specified intervals of time. For example, the transfer of the value of the storage element a to the element b

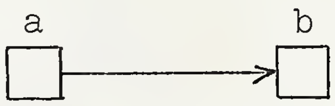


Fig. 4.10.1

is valid at times $t+\varepsilon_o, t+\varepsilon_1, \ldots$ . The elements of a circuit are activated simultaneously by a timing circuit (not shown) and each activation is called a "shift." The values of a and b may be thought of as Boolean functions of time

$$b(t+\varepsilon_i) = a(t+\varepsilon_{i-1}), i= 1,2,\ldots$$

so that the circuit

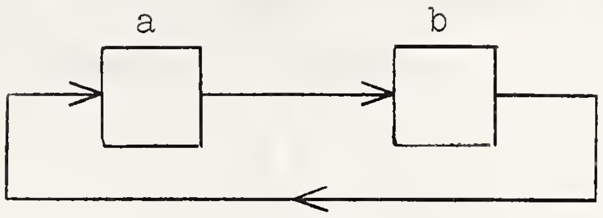


Fig. 4.10.2

represents a recirculation of the values of a and b, i.e.,

$$b(t+\varepsilon_i) = a(t+\varepsilon_{i-1})$$
$$a(t+\varepsilon_i) = b(t+\varepsilon_{i-1}) \quad i=1,2,\ldots \quad .$$

For example, if $a(t+\varepsilon_o) = 0$ and $b(t+\varepsilon_o) = 1$, the successive values of a and b represent the sequence of vectors

$$\begin{matrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ & \vdots \end{matrix}$$

If we include a modulo 2 adder in the recirculating circuit as in Fig. 4.10.3

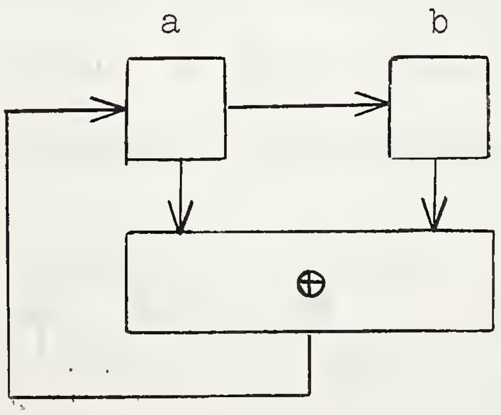


Fig. 4.10.3

and start with the vector (a,b) = (0,1), the circuit "generates" the sequence of vectors

$$\begin{matrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ & \vdots \end{matrix}$$

The circuit has generated a cyclic sequence of 3 distinct vectors and could be used, for example, to count modulo 3.

This type of circuit and the sequences of vectors they generate have been studied extensively by Elspas, Huffman,[1], Zierler and Prange. The circuits are sometimes called binary sequence generators for obvious reasons. Another term which we shall use is "feedback shift registers." The elements a,b may be regarded as a "register" which will hold 2 elements whose contents are shifted "right" while a digit is fed back into the left hand end of the register. In what follows we shall conserve space by suppressing the connections between the elements of a register as in Fig. 4. 10.4.

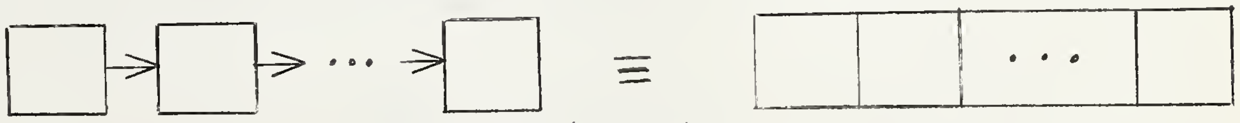


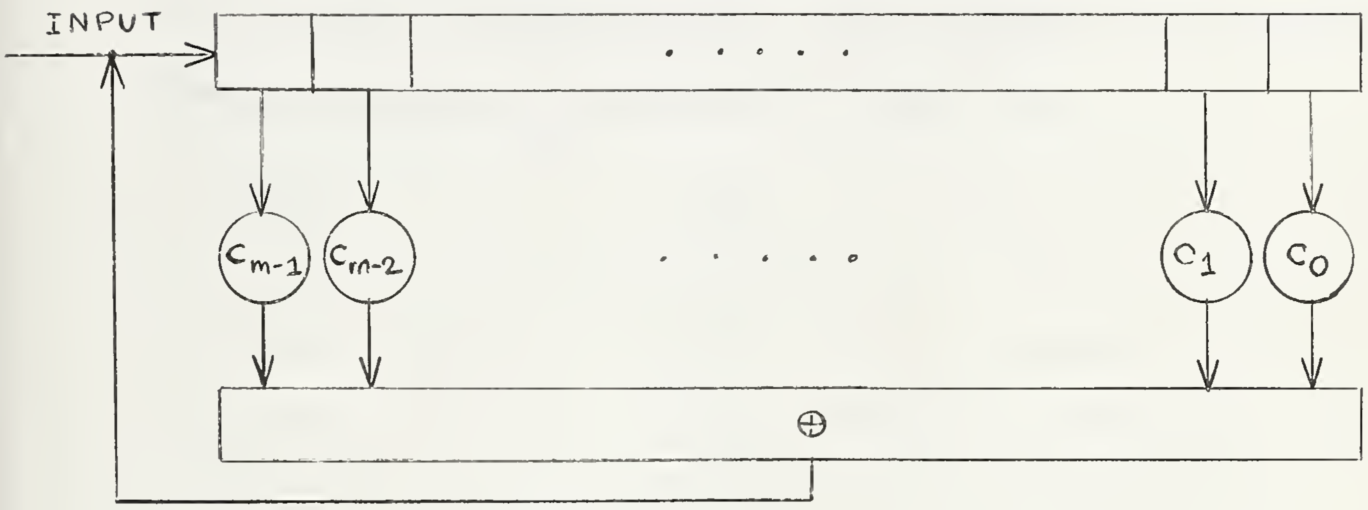
Fig. 4.10.4

## Section 4.11 Matrices, Feedback Shift Registers, and Cyclic Codes

The structure and operation of feedback shift registers can be described by matrices whose elements are 0 or 1 and whose operators are " $\oplus$ " and ".". The matrices corresponding to feedback shift registers are exactly the T matrices of section 4.9 and the circuit corresponding to a matrix whose characteristic equation is

$$T^m+c_{m-1}T^{n-1}+\ldots+c_1T+c_o = 0$$

is shown in Fig. 4.11.1



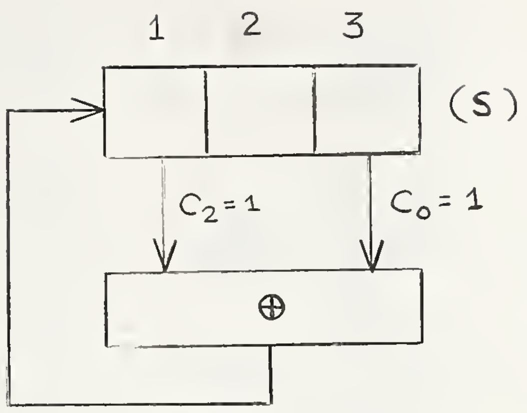
For example, the circuit corresponding to the matrix

$$T = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

whose characteristic equation is

$$T^3 + T^2 + 1 = 0$$

is



Consider what happens if the register s initially contains 100 in positions 1,2,3 respectively, and the circuit is activated. At each activation the contents of 1 replace the contents of 2, the contents of 2 replace the contents of 3 and the sum modulo 2 of 1 and 3 replace the contents of 1. Then the successive contents of s will be

```
1 0 0
1 1 0
1 1 1
0 1 1
1 0 1
0 1 0
0 0 1
1 0 0
  :
```

After 7 shifts the original vector 100 reappears in s. The vectors generated by the circuit are precisely those obtained by pre-multiplying the column vector (1,0,0) successively by T.

$$s = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad Ts = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad T^2s = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$T^3s = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad T^4s = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \quad T^5s = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$T^6s = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad T^7s = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} .$$

It is easy to show directly that $T^7 = I$ if T satisfies

$$T^3 + T^2 + 1 = 0.$$

For an m x m matrix T, the original vector will reappear after $2^m$-1 multiplications or less. If the original vector reappears after $2^m$-1 shifts, the sequence of vectors generated is called maximal length sequence. The 3 x 3 matrix in the example generates a maximal length sequence of $2^3$-1 = 7 vectors. Of course, if the vector s is the zero vector the length of the sequence will be 1 for everymatrix T. As we shall see, the length of the sequence generated by a m x m matrix T defines n, the length of a cyclic code, where m of these n digits are parity checks. Hence, for a given m, the larger the sequence of vectors is, the larger the number of distinct vectors in the code. In the example, the 3 x 3 matrix defines a (7,4) code which has $2^4$ code vectors.

## Section 4.12 Encoding of Cyclic Codes

To define codes generated by feedback shift registers it is again simplest to describe an encoder and show that it generates a cyclic code whose vectors $x = (x_1,\ldots,x_n)$ satisfy

$$x_1 T^{n-1} s_1 \oplus x_2 T^{n-2} s_1 \oplus \ldots \oplus x_n s_1 = 0.$$

Rather than describe a general encoder, we may employ the circuit of the preceding section as an encoder; the generalisation will be immediately obvious. Only a few more connections and one modulo 2 adder need to be added to the feedback shift register to construct an encoder

(4)

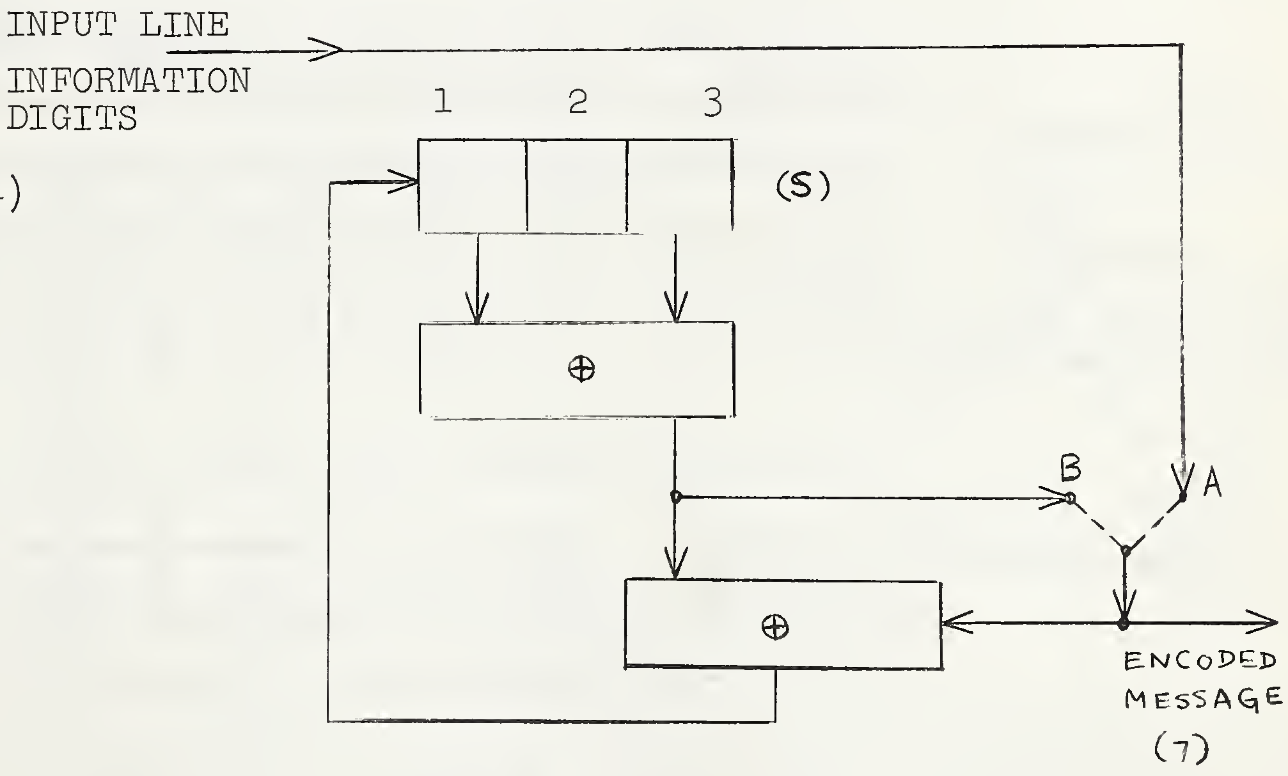


Fig. 4.12.1

The switch is in position A while the information

digits are arriving (4 shifts) and then in position B while the check digits which were formed in the register from the information digits are being read out (3 shifts). The encoded message then will consist of 4 information digits followed by 3 check digits. Initially the register contains 0 0 0. The sequence of events may be easily followed if we encode the information digits $x = (x_1,x_2,x_3,x_4) = (0,1,0,1)$. $x_1$ enters first. The first shift puts $x_1 = 0$ on the output line and $x_1s_1$ in the register s where $s_1 = (1,0,0)$. The second shift puts $x_2 = 1$ on the output line, and $x_1Ts_1 \oplus x_2s_1$ in the register, and so on. The sequence of events may be shown in the following diagram.

| Digit In | No. of Shifts | 1 | 2 | 3 | Digit OUT | Contents of s |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | | |
| 0 | 1 | 0 | 0 | 0 | 0 | $x_1s_1$ |
| 1 | 2 | 1 | 0 | 0 | 1 | $x_1Ts_1 \oplus x_2s_1$ |
| 0 | 3 | 1 | 1 | 0 | 0 | $x_1T^2s_1 \oplus x_2Ts_1 \oplus x_3s_1$ |
| 1 | 4 | 0 | 1 | 1 | 1 | $x_1T^3s_1 1\oplus x_2T^2s_1 \oplus x_3Ts_1 \oplus x_4s_1$ |
| | 5 | 0 | 0 | 1 | $x_5 = 1$ | $x_1T^4s_1\oplus\ldots\oplus x_5s_1$ |
| | 6 | 0 | 0 | 0 | $x_6 = 1$ | $x_1T^5s_1\oplus\ldots\oplus x_6s_1$ |
| | 7 | 0 | 0 | 0 | $x_7 = 0$ | |

Fig. 4.12.2

At the end of 7 shifts the register s contains (0,0,0).

Hence $x_1T^6s_1 + x_2T^5s_1 + \dots + x_6Ts_1 + x_7s_1 = 0$ which is the condition a (7,4) cyclic code should satisfy. To check that $x = (x_1, x_2, \dots, x_7) = (0101110)$ does satisfy this vector equation, we note that

$$0.\begin{bmatrix}0\\0\\1\end{bmatrix} \oplus 1.\begin{bmatrix}0\\1\\0\end{bmatrix} \oplus 0.\begin{bmatrix}1\\0\\1\end{bmatrix} \oplus 1.\begin{bmatrix}0\\1\\1\end{bmatrix} \oplus 1.\begin{bmatrix}1\\1\\1\end{bmatrix} \oplus 1.\begin{bmatrix}1\\1\\0\end{bmatrix} \oplus 0.\begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix}$$

The encoder uses the following parity check equations

$$x_{i+1} \oplus x_{i+3} \oplus x_{i+4} \oplus x_{i+5} = 0, \; i=0,1,2$$

## Section 4.13 Decoding of Cyclic Codes

The code generated by this encoder is a single-error-correcting code and a decoder could be devised based on the standard coset array developed earlier. However, there is a much simpler type of decoding apparatus available which follows directly from the method of encoding. Again, for simplicity we will describe the special case of $m = 3$ and indicate the generalisation. The decoder for our (7,4) code with characteristic equation $T^3+T^2+1 = 0$ has two registers - one a three-digit feedback register with connections similar to those for the encoder and the other a 7-digit shift register which holds the received vector until it can be corrected.

The decoder has a detector which is a circuit which will emit a 1 when a certain configuration of digits occurs in the feedback register. The detector is switched off while the 7 message digits are being received and the check digits recalculated by the feedback circuit. The detector is switched on and the shifting continues and digits are sent to the output line. The detector is designed so that when an incorrect digit reaches the right-hand end of the 7 digit register the detector will emit a 1 which will be added modulo 2 to the erroneous digit, correcting it.

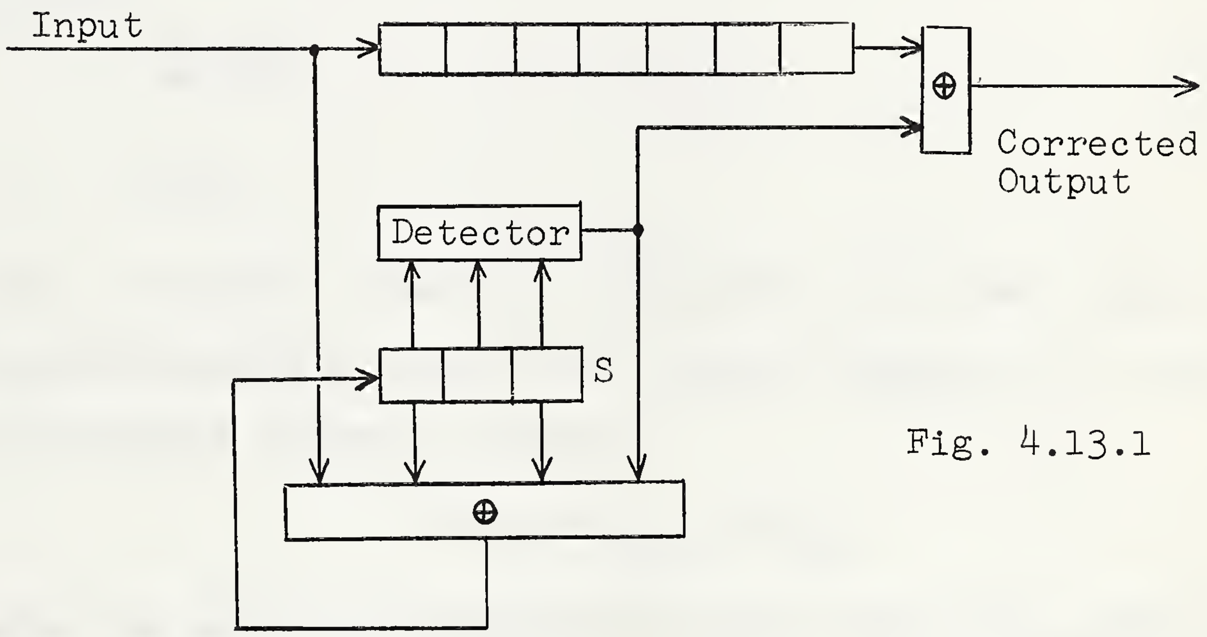


Fig. 4.13.1

If the vector x is processed through a general decoder of this type, after all n digits, $x_1, \dots, x_n$ have been received the register S will contain

$$x_1'T^{n-1}s_1 \oplus x_2'T^{n-2}s_1 \oplus \ldots \oplus x_1's_1 = z \qquad \ldots\ldots 4.13.1$$

and if there is a single error in $x'$ then z will not be zero.

If there is a single error in position r, $x_r' = x_r \oplus 1$. Since $T^n = I$, Equation 4.13.1 may be written

$$z = x_1'T^{-1}s_1 \oplus x_2'T^{-2}s_1 \oplus \ldots \oplus x_r'T^{-r}s_1 \oplus \ldots \oplus x_n'T^{-n}s.$$

$$= (x_1T^{-1}s_1 \oplus x_2T^{-2}s_1 \oplus \ldots \oplus x_rT^{-r}s_1 \oplus \ldots \oplus x_nT^{-n}s_1)$$

$$\oplus T^{-r}s_1$$

$$= T^{-r}s_1.$$

After r-1 shifts following the reception of the n digits, the erroneous digit $x_r'$ leaves the n-digit register. At that time the feedback register contains

$$T^{r-1}(T^{-r})s_1 = T^{-1}s_1.$$

Hence the detector must recognize the state $T^{-1}s_1$ which is (0,...,0,1) since $s_1 = (1,0,\ldots)$ for all T. Hence, for single error correction, the detector may be a very simple circuit. When the detector recognizes the state $T^{-1}s_1$, it emits a 1 which is added modulo 2 to the rth digit which is ermerging from the main register. Thus the error is corrected.

The operation of the decoder for our (7,4) may be illustrated by the processing of the vector 0001110 which has an error in the second position.

| Digit In | No. of Shifts | Main Register | | | | | | | Feedback Register | | | Digit Out |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 0 | 1 | 0 | | | | | | | 0 | 0 | 0 | |
| 0 | 2 | 0 | 0 | | | | | | 0 | 0 | 0 | |
| 0 | 3 | 0 | 0 | 0 | | | | | 0 | 0 | 0 | |
| 1 | 4 | 1 | 0 | 0 | 0 | | | | 1 | 0 | 0 | |
| 1 | 5 | 1 | 1 | 0 | 0 | 0 | | | 0 | 1 | 0 | |
| 1 | 6 | 1 | 1 | 1 | 0 | 0 | 0 | | 1 | 0 | 1 | |
| 0 | 7 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | |
| Detector On | 8 | | 0 | 1 | 1 | 1 | 0 | $\boxed{0}$ | 0 | 0 | 1 | 0 |
| | 9 | | | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | $\boxed{1}$ |
| | 10 | | | | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 11 | | | | | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| | 12 | | | | | | 0 | 1 | 0 | 0 | 0 | 1 |
| | 13 | | | | | | | 0 | 0 | 0 | 0 | 1 |
| | 14 | | | | | | | | 0 | 0 | 0 | 0 |

Fig. 4.13.2

The detector also emits a 1 into the feedback register to prevent further error correction taking place.

The cyclic codes constitute a very important class of codes and many powerful, easily implemented cyclic codes

are known. They are particularly useful in correcting "burst" errors. Burst errors may be described by their error vectors, e.g., a burst of "length" 3 would be one of the error patterns, 111,101,100,110,011,001,010, located anywhere in a vector. These are important types of errors because in many channels the presence of one error makes it likely that the digits adjacent to it are also in error. The discovery of codes with burst-correcting properties depends on finding suitable irreducible polynomials which are the subject of certain topics in modern algebra. A large selection of irreducible polynomials is listed in Peterson p. 251-270.

## Section 4.14 Conclusion

The codes we have presented represent some of the main developments of coding theory. A great many codes are available for application to computers yet at the present time only a relatively few are in use (Buchholz; Dimsdale and Weinberg; Honeywell; IBM [1]). There are many reasons for this. One of the more important is that computers still operate in a somewhat pampered environment and it is still possible to design channels conservatively enough to equal the error-correcting abilities of short codes. However, it appears that the time is fast approaching when computers will be required to operate unattended in noisy environments,

both external and internal, for which conventional methods will be sufficiently unreliable that error correction is necessary. The short-sequence lengths demanded by present computers confine the choice of codes to the least efficient types. Despite the large number of elements in a computer, e.g., storage elements, error-correcting methods have to be applied to small sets of elements independently; hence there is no way of taking advantage of the possibility of using highly efficient "long" codes.

Despite all of these problems, it is likely that computer technology will benefit eventually. These codes are very important and practical for communication systems and as experience is gained with them in this field, it is likely to be applied to computers, since computer technology and communication engineering are closely related.

## Appendix 4a. Groups, Fields and Vector Spaces.

### Groups.

The binary alphabet consists of the set of two elements, called 0 and 1 for convenience. We are interested in showing that {0,1} satisfy the axioms for a group under the operator + defined below:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

.

The axioms for a group are

1) Closure: The operation applied to any two elements gives another element of the group. By inspection this is satisfied.

2) Associativity: For any 3 elements a,b,c of the group (a+b)+c = a+(b+c). By inspection of all 8 possible situations, this is satisfied.

   E.g., (0+0)+1 = 0+1 = 1

   0+(0+1) = 0+1 = 1 .

3) Existence of Identity Element Identity element satisfies 0+a = a+0 = a. Since 0+1 = 1+0 = 1 and 0+0 = 0+0 = 0, the element 0 is the identity element.

4) Existence of Inverses Every element of group possesses an inverse within the group. i.e., for every element a, an inverse (-a) exists satisfying (-a)+a = a+(-a)=0.

Since 0+0 = 0 and 1+1 = 0, each element is its own inverse.

Further, the group {0,1} is commutative under + since 0+1 = 1+0.

## Fields

In order to justify the use of the linear algebra used in the chapter, we must show that {0,1} form a field under the operations

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Addition

| . | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Multiplication

A field is a commutative ring with a multiplicative identity in which every non-zero element has a multiplicative inverse. The axioms for a commutative ring are proved first.

They are

1) The set of elements form a commutative group under addition. This was proved earlier.
2) Closure under multiplication. For any two elements a,b the product a.b is an element of the ring. This is shown by inspection of the table.
3) Associativity: a(bc) = (ab)c. This can be shown

by exhibiting all 8 possibilities.

4) Distributivity: For any 3 elements, a(b+c) = ab+ac and (b+c)a = ba+ca. Since multiplication is commutative (by inspection) we need only prove one of these. a(b+c) = ab+ac holds for all 8 possibilities.

Since the only non-zero element, 1, is its own multiplicative inverse, the set $\{0,1\}$ forms a field.

## Vector Spaces

A set V of elements is a vector space over a field F if it satisfies the axioms,

1) The set V is a commutative group under addition.
2) For any vector v and any field element c, a product cv is defined and is a vector.
3) If u and v are vectors of V and c is a field element c(u+v) = cu+cv.
4) If v is a vector and c and d are field elements (c+d)v = cv+dv.
5) If v is a vector and c and d are scalars (cd)v = c(dv), and 1v = v.
(The multiplicative operator is indicated by juxtaposition.)

The vectors we are interested in are sequences of field elements 0 and 1.

Addition of vectors is defined element by element, viz. $(a_1,a_2,\dots,a_n)+(b_1,b_2,\dots,b_n)$

$$= (a_1+b_1,a_2+b_2,\dots,a_n+b_n).$$

Since $a_i$ and $b_i$ are field elements, $a_i+b_i$ are field elements and the addition of two vectors defines another vector. It can be shown by similar reasoning from the element-by-element definition that the vectors form a group under +. The identity element is $(0,0,\dots,0)$.

Multiplication of a vector by a field element is also defined term by term

$$c(a_1,a_2,\dots,a_n) = (ca_1,ca_2,\dots,ca_n).$$

The result is a vector since $ca_i$ are field elements, hence 2) is satisfied. 3),4) and 5) can be shown to hold by exhibiting the structure of the vectors in terms of field elements.

Finally, the dot product of two vectors may be defined as

$$(a_1,a_2,\dots,a_n).(b_1,b_2,\dots,b_n)$$

$$= a_1b_1+a_2b_2+\dots+a_nb_n$$

which is a field element.

These definitions suffice to justify the use of the matrix operations throughout the chapter. Further development of the cyclic codes introduced in section 4.9 require the concepts of polynomial rings and Galois fields. See Peterson, Chapter 6.

## Appendix 4b.

We quote a few results adapted from those given in Peterson (Chapter 4.)

1) The Plotkin Bound

The minimum weight of a code vector in an $(n,k)$ linear code is at most $n2^{k-1}/(2^k-1)$.

If $B(n,d)$ is the maximum number of code vectors possible in a linear code of length n with minimum weight at least d, then for $n>d$

$$B(n,d) \leq 2B(n-1,d).$$

These two results may be combined to give

$$B(n,d) \leq d.2^{n-2d+2}$$

and the Plotkin bound is

$$\underline{k \leq n - 2d+2+\log_2 d} \ .$$

2) Hamming Bound (See Section 4.3)

Any n-digit code (linear or non-linear) with minimum weight at least $2m+1$ must have at least

$\log_2 [1+\binom{n}{1}+\binom{n}{2}+\ldots+\binom{n}{m}]$ check symbols.

For $n \to \infty$, it can be shown that

$$1-\frac{k}{n} \leq H(\frac{m}{n})$$

where H(x) is the entropy function of Chapter 3.

3) Varsharmov - Gilbert Bound

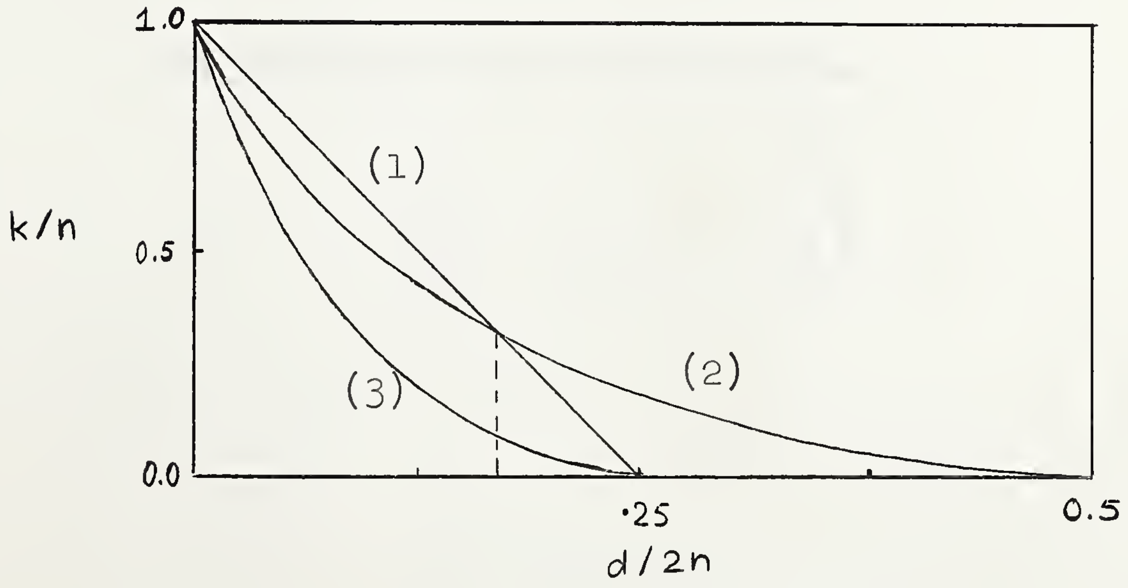
It is possible to construct a code of length n and minimum distance d with r parity check digits where r is the smallest integer satisfying

$$\binom{n-1}{1} + \binom{n-1}{2} + \ldots + \binom{n-1}{d-2} < 2^r - 1$$

For n sufficiently large

$$1 - \frac{k}{n} \geq H\left(\frac{d}{n}\right).$$

The first two are upper bounds, the third a lower bound. As n becomes large, if the rate k/n is kept constant, the bounds on the ratio d/n become fixed constants independent of n. They are plotted below for "best" codes (See Appendix 4c.)



[illegible]

## Appendix 4c. Parity Check matrices for best linear codes.

A "best" or optimum linear code is a code for which the probability of error is as small as for any other linear code with the same n and k. The channel is assumed to be the binary symmetric channel.

The table lists parity check matrices for values of n up to 9 and k up to 7. The use of the table will be indicated with an example. The matrix for the (5,3) code reads

$$\begin{array}{ccc} 4 & 1 & 2 \\ 5 & 1 & 3 \end{array} .$$

This corresponds to the parity check equations

$$x_4 \oplus x_1 \oplus x_2 = 0$$
$$x_5 \oplus x_1 \oplus x_3 = 0.$$

The corresponding matrix is thus

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

|  | k = 2 | k = 3 | k = 4 | k = 5 | k = 6 | k = 7 |
|---|---|---|---|---|---|---|
| n = 4 | 3 2 |  |  |  |  |  |
|  | 4 1 2 |  |  |  |  |  |
| n = 5 | 3 1 2 | 4 1 2 |  |  |  |  |
|  | 4 2 | 5 1 3 |  |  |  |  |
|  | 5 1 |  |  |  |  |  |
| n = 6 | 3 2 | 4 1 2 | 5 1 2 3 |  |  |  |
|  | 4 1 2 | 5 1 3 | 6 1 2 4 |  |  |  |
|  | 5 1 | 6 2 3 |  |  |  |  |
|  | 6 1 |  |  |  |  |  |
| n = 7 | 3 1 | 4 1 3 | 5 1 3 4 | 6 1 |  |  |
|  | 4 1 | 5 1 2 | 6 1 2 4 | 7 1 |  |  |
|  | 5 1 | 6 1 2 3 | 7 1 2 3 |  |  |  |
|  | 6 1 2 | 7 1 2 3 |  |  |  |  |
|  | 7 2 |  |  |  |  |  |
| n = 8 | 3 1 | 4 1 | 5 1 3 4 | 6 1 3 4 | 7 1 |  |
|  | 4 1 | 5 1 2 | 6 1 2 4 | 7 1 2 4 | 8 1 |  |
|  | 5 2 | 6 1 3 | 7 1 2 3 | 8 1 2 3 |  |  |
|  | 6 2 | 7 2 3 | 8 1 2 3 4 |  |  |  |
|  | 7 1 2 | 8 1 2 3 |  |  |  |  |
|  | 8 1 2 |  |  |  |  |  |
| n = 9 | 3 1 | 4 1 | 5 1 3 4 | 6 1 3 4 5 | 7 1 3 4 | 8 1 |
|  | 4 1 | 5 2 | 6 1 2 4 | 7 1 2 4 5 | 8 1 2 4 | 9 1 |
|  | 5 1 | 6 1 2 | 7 1 2 3 | 8 1 2 3 5 | 9 1 2 3 |  |
|  | 6 2 | 7 1 3 | 8 1 2 3 | 9 1 2 3 4 |  |  |
|  | 7 2 | 8 2 3 | 9 1 2 3 |  |  |  |
|  | 8 1 2 | 9 1 2 3 |  |  |  |  |
|  | 9 1 2 |  |  |  |  |  |

Table 4c.

# CHAPTER 5

## The Structure of a Digital Computer

This chapter presents certain difficulties and it is suggested that the reader examine first the main points of the chapter by reading Sections 5.1, 5.2, and 5.5 together with Fig. 5.3.1 and then return to the rest of the chapter.

The difficulties arise because of the large amount of detailed information presented and the relatively unfamiliar notation used. The detailed information is hardly avoidable but the introduction of a complicated notation requires some justification. In this chapter, the notation has four main requirements to satisfy: it must be precise; it must be concise because of the volume of information it must convey; it must take into account the sequential nature of the processes to be described; it must permit easy manipulation of operands which have the structure of vectors and matrices.

The first two requirements may be met by the careful use of well-defined short symbols which represent operands; this suggests the use of existing mathematical notations. A good mathematical notation is usually a first step to understanding and generalisation; (desirable traits which are very clearly lacking in the digital computer field). The third requirement is easily satisfied by numbering statements made. The fourth presents difficulties which are not easily overcome. While many of the operations to be performed on vectors and matrices are the usual ones for which symbols already exist in various mathematical fields, many of the manipulations which occur frequently enough to require concise symbols do not appear explicitly elsewhere, except perhaps verbally. In addition, all of the operators may appear simultaneously in an algorithm (a sequence of statements) and, consequently, must have different symbols. The language employed in this chapter makes a great deal of use of special constant vectors to aid in making the notation as concise as possible while permitting a wide range of well-defined manipulations not normally expressed in mathematical form. All of these contribute to the unusual appearance of algorithms in this notation and make the language a difficult one on a first approach. However, to the writer at least, this is indeed a small price considering the very wide range of applications of the notation. In particular, it may be applied to the whole range of problems associated with computers, from numerical analysis to logical design and is presented here with that in mind.

## Section 5.1 Introductory Remarks

The subject matter of this chapter is, on the surface at least, quite different from that of the two preceding chapters. There are definite connections between these fields, of course, and there can be no doubt that the interactions between them will increase greatly in the near future. Two connections are immediately evident - the information handled by computers is precisely the information that information theory deals with and a digital computer may be regarded as a very complex channel or a complex of interrelated binary channels.

It is also evident that computers are constructed from the same basic elements as the encoders and decoders of Chapter 4. Unfortunately, these connections are rendered almost vacuous by the remarkable overall complexity of the digital computer and it is not yet possible to carry over the methods of these fields into the study of computers. Information theory has scarcely begun to consider the problems related to the study of two interconnected channels whereas there may be ten or twenty channels in operation simultaneously in a digital computer. Also the encoders and decoders for cyclic codes of the last chapter could be constructed from a few hundred binary elements while the number of elements in a reasonably powerful computer is

several orders of magnitude greater. In the face of this complexity, it is a difficult problem even to describe a digital computer without degenerating into vagueness or getting lost in extreme detail.

This chapter is an attempt to describe a digital computer the IBM 1620, at a level somewhere between a description of what it can do and a description of its circuitry. Both of these descriptions are useful but neither contributes much to an understanding of how digital computers fit into the context of information theory and coding theory. Clearly, it is essential to know what tasks a computer is to perform and what basic physical devices are available for its construction. However, with some reasonable information on both of these, such as a description of the language the computer uses and idealised models of the physical components available, it is possible to suppress "irrelevant" detail in both of these areas and arrive at a satisfactory model of the computer. Certain aspects of this model can be studied with such tools as Boolean algebra, graph theory and algorithmic languages but an understanding of the overall structure of the model lies beyond the scope of the present mathematical methods.

The particular aspect of computers we shall explore

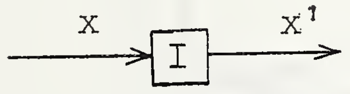
in this chapter is often called the structural organisation of computers. It offers, in the first instance, a reasonably comprehensive viewpoint on the detailed operation of machines and, secondly, is amenable to a primarily mathematical description. Further, this area has received much more attention lately because of certain developments in the fields of programming languages and of circuitry and components. Languages such as FORTRAN and Algol have become indispensable in the efficient statement of algorithms to be executed by machines but these languages often are ill-adapted to translation into the present machine languages. Radically different structural organisations are required to solve this problem and a few such computers have appeared. At the other extreme, it has been possible so far to keep the overall performance of a machine at a satisfactory level by increasing the speed of the basic elements, i.e., the time it takes to switch from one state to another, but improvement in this direction is limited eventually by the time it takes for an electrical pulse to travel from one place to another within the machine. Here again, the structural organisation will have to be modified to overcome this problem.

## Section 5.2 Basic Machines

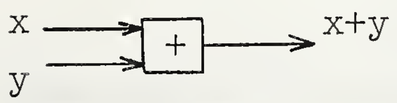
The word "machine" is here used in a wide sense.

It designates any rational collection of the elementary binary devices which will be described shortly. While a digital computer is a machine itself, it is more easily studied as a collection of elementary machines.

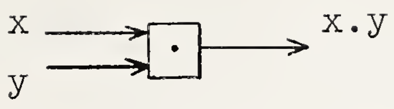
In Chapter 4 we examined some of the elementary devices that are used in the construction of feedback shift registers. These devices are binary storage elements and the hardware realisations of the operations " ⊕ " and "." . Two other binary elements are commonly used in the construction of machines; they are "I", an inverter and "+", the OR operation. These elements may be described in terms of Boolean algebra where the binary storage element can be thought of as having the properties of a Boolean variable which can take on the values 0 or 1 and the operations are Boolean operations. The operations are defined by the tables in Fig. 5.2.1 and are shown with their corresponding schematic diagrams.



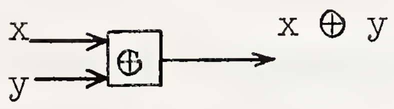
| x | x' |
|---|---|
| 0 | 1 |
| 1 | 0 |



| x | y | x+y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



| x | y | x.y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



| x | y | x⊕y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Fig. 5.2.1

The ⊕ operation is physically realized in terms of the others as xy' + x'y (Fig. 5.2.2) but is used so often that it is convenient to define it as a basic operation. The equivalent circuit

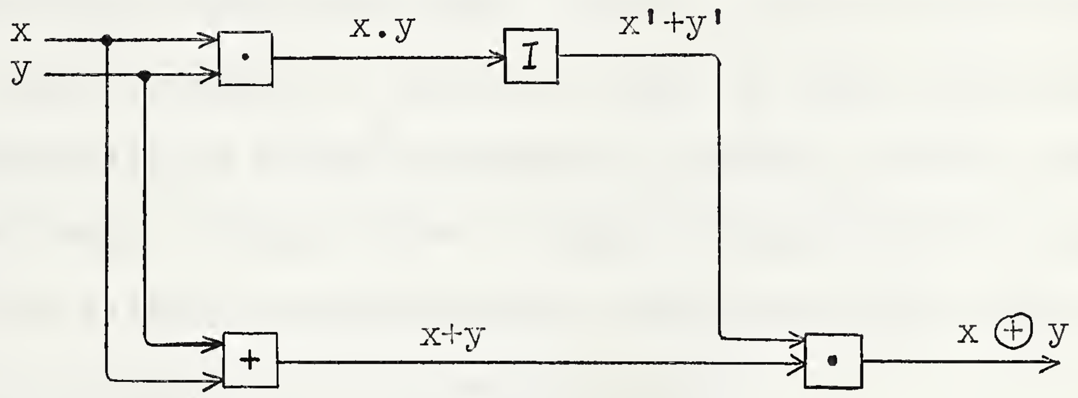


Fig. 5.2.2

makes use of the well-known DeMorgan's theorems of Boolean algebra;

$$(x+y)' = x'.y'$$

[illegible]

[illegible]

[illegible]

[illegible]

and

$$(x.y)' + x'+y'.$$

## Section 5.2 Limitations of Boolean Description

It is important to note the limitations of a Boolean description of physical circuits. The value or state of a device is valid only at certain intervals of time - at the moment of changing from one value to another, the value is clearly undefined. In a physical device, there is actually an interval of time, rather than an instant, during which the values are undefined. Hence Boolean algebra can only describe the steady states of binary devices.

Further, the circuits corresponding to operations (usually called "gates") hold the results of an operation only for a very short time. In Fig. 5.2.2, there should be a "delay" element to hold the value of (x+y) while the I-operation is being performed. However, we will assume that these and most other timing problems are the province of the circuit designer while taking note that they exist as a limitation of our Boolean model.

The only other limitation of note is the fact that there are certain probabilities of crossovers in the binary elements. In other words, each element is a binary

noisy channel (most likely asymmetric.) The description of a probabilistic scheme is clearly beyond the scope of Boolean algebra. However, encoders and decoders which could be included in a large machine to increase the probability of correct transmission can be described in Boolean terms.

With these reservations in mind we may apply the methods of Boolean algebra to the description of machines.

## Section 5.2.2 Transfers between Registers.

The circuit of Fig. 5.2.2 calculates the value of the Boolean function $f(x,y) = x'.y+y'.x$ given the values of the independent variables x and y. Normally, the values of x and y would be stored in two binary storage elements and the result of the calculation of f(x,y) would be transmitted to another storage element for later use. The operation of the circuit can be regarded as a transfer of information from the two storage elements holding the values x and y to another storage element. If we use the name of the variable to denote the storage element whose state represents the variable, the transfer of information between elements can be written as

$$z \leftarrow f(x,y) \ldots\ldots\ldots \qquad 5.2.1$$

where the value that is transferred to the storage element

z is defined by the Boolean equation f(x,y) = x'.y+y'.x. The physical circuits which perform the calculation take a certain length of time to operate so that the element z can be said to contain the result of the calculation only after this time has elapsed. Hence Equation 5.2.1 is a shorthand notation for the Boolean equation

$$z(t+\varepsilon) = f(x(t),y(t))$$

where t is the time at which the circuit is activated and ε is the time it takes the circuit to reach its steady state. It is also possible to specify meaningful transfers of the type

$$z \leftarrow f(z,y) \quad \ldots\ldots\ldots\ldots \quad 5.2.2$$

since this represents

$$z(t+\varepsilon) \leftarrow f(z(t),y(t)).$$

Expressions such as 5.2.1, 5.2.2 are often called statements to avoid the mathematical connotation of "equations."

It should be noted, also, that the above use of x,y,z to denote both the value stored in an element and the element itself need not cause confusion. When the operator "←" appears in an expression,x,y,z should be read as "the contents of x, the contents of y, etc." The symbol "=",

however, appears in statements occasionally as a Boolean operator.

The transfer notation may be easily extended to the concept of registers. A register is defined as an ordered set of binary storage elements. The ordering permits a useful identification of the contents of the register with a positional number system. For example, a five-binary-digit register may represent the integers 0 through 31 by associating its elements with the set of weights (16,8,4, 2,1). Again, a ten binary-digit register may represent the integers 0 through 165 when its elements correspond to the weights (0,80,40,20,10,0,8,4,2,1). (The elements of the register which correspond to zero weights are usually parity digits.)

It is convenient to order the elements in a register using vector notation e.g., the n-element register $\underline{x}$ comprises the elements $\underline{x}_0, \underline{x}_1, \underline{x}_2, \ldots, \underline{x}_{n-1}$. Index origins other than 0 may be used but unless otherwise explicitly stated 0-origin indexing will be used in what follows. The statement

$$\underline{z} \leftarrow f(\underline{x}, \underline{y})$$

means that corresponding elements of the n-digit registers $\underline{x} = (\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_{n-1})$, $= (\underline{y}_0, \underline{y}_1, \ldots, \underline{y}_{n-1})$ are used as

arguments of the function f and the results transferred to the corresponding elements of the n-digit register $\underline{z}$, i.e.,

$$\underline{z} \leftarrow f(\underline{x},\underline{y}) \text{ implies } \underline{z}_i \leftarrow f(\underline{x}_i,\underline{y}_i) \quad i = 0,1,\ldots,n-1.$$

Such transfers are not defined unless the registers are compatible, i.e., of the same length.

The concept of transfers between registers is a powerful tool for understanding how a sequential machine functions and it is well worth the effort to find a notation which facilitates the description of transfers. Such a notation or language has many potential applications where-ever well-defined sequential process are encountered, (Iverson [2]).

## Section 5.2.3 Notation for the Description of Transfers

The notation we shall adopt is that of Iverson [1], and is best introduced by an example illustrating its use. Two of the important registers of the IBM 1620 are called the Memory Data Register and the Operation Register. Let us denote the former, a 6-binary-digit register, by $\underline{d} = (\underline{d}_0,\underline{d}_1,\underline{d}_2,\underline{d}_3,\underline{d}_4,\underline{d}_5)$ and the latter, a 10-binary-digit register, by $\underline{o} = (\underline{o}_0,\underline{o}_1,\ldots,\underline{o}_9)$. The digit $\underline{d}_0$ represents a parity-check digit for the other digits in $\underline{d}$. It is desired to transfer the digits in $\underline{d}_2,\underline{d}_3,\underline{d}_4,\underline{d}_5$ to $\underline{o}_1,\underline{o}_2,\underline{o}_3,\underline{o}_4$ and to

transfer to $\underline{o}_1$ the parity-check digit for $\underline{d}_2,\underline{d}_3,\underline{d}_4,\underline{d}_5$. The digit in $\underline{d}_1$ is not to be transferred to $\underline{o}$. The digit-by-digit transfers are;

$$\boxed{\begin{array}{l} \underline{o}_1 \leftarrow \underline{d}_2 \\ \underline{o}_2 \leftarrow \underline{d}_3 \\ \underline{o}_3 \leftarrow \underline{d}_4 \\ \underline{o}_4 \leftarrow \underline{d}_5 \\ \underline{o}_0 \leftarrow (\underline{d}_0 \neq \underline{d}_1) \end{array}} \quad \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}$$

Fig. 5.2.4

The meaning of lines 1 through 4 is obvious. Line 5 represents the calculation of the new parity digit from the old parity digit in $\underline{d}_0$ and the digit in $\underline{d}_1$. It may be explained as follows; the digit in $\underline{d}_1$ is to be dropped from the parity check and this can be done by subtracting the digit in $\underline{d}_1$ modulo 2 from the parity check digit. Line 5 could be written as

$$\underline{o}_0 \leftarrow \underline{d}_0 \oplus \underline{d}_1$$

since modulo 2 addition and subtraction are identical. However it happens that the relational operator $\neq$ used in Iverson's notation is identical with the exclusive-or operation when its operands are restricted to the values 0 and 1.

[illegible]

In general, the relation (xRy) has the value 1 when the relation is satisfied and the value 0 when the relation is not satisfied. (R is any relational operator and x and y are any pair of variables of the same type). In the special case of ($\underline{d}_0 \neq \underline{d}_1$) we may form the table:

| $\underline{d}_0$ | $\underline{d}_1$ | $\underline{d}_0 \neq \underline{d}_1$ | $\underline{d}_0 \oplus \underline{d}_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

Table 5.2.1

It is clearly desirable to express the program in Fig. 5.2.4 more compactly. This can be done in a number of ways, one of which is to make use of the "selection" operator "/". The selection is controlled by a logical vector of 1's and 0's which has 1's in the positions to be selected. The result of the selection is a vector whose dimension is the number of 1's in the logical vector, e.g., for $\underline{u} = (1,0,0,1,1)$, $\underline{k} = (1,2,-3,-9,4)$

$$\underline{u}/\underline{k} = (1,-9,4).$$

Further, a number of standard vectors have been defined to aid in manipulating vectors. We shall use

$$\underline{\alpha}^j(n) = (\underbrace{1,1,\ldots,1}_{j},0,0,0,\ldots,0)$$

$$\overset{n}{\phantom{x}}$$

$$\underline{\omega}^j(n) = (0,0,0,\ldots,0,\underbrace{1,1,\ldots,1}_{j})$$

$$\overset{n}{\phantom{x}}$$

and $$\underline{\varepsilon}(n) = (\underbrace{1,1,1,\ldots,1}_{n})$$

In particular $\underline{\alpha}^5(10) = (1,1,1,1,1,0,0,0,0,0)$ and $\underline{\omega}^4(5) = (0,1,1,1,1)$. Program 5.2.4 may now be written as

$$\underline{\alpha}^5/\underline{o} \leftarrow (\underline{d}_0 \neq \underline{d}_1),(\underline{\omega}^4/\underline{d})\ldots \qquad 5.2.5$$

The dimensions of $\underline{\alpha}^5$ and $\underline{\omega}^4$ are elided since they would be clear from the context of a larger program in which statement 5.2.4 would appear. On the right hand side, $\underline{\omega}^4/\underline{d}$ selects the last four digits of $\underline{d}$ viz., $\underline{d}_1,\underline{d}_2,\underline{d}_3,\underline{d}_4$ and catenates them (",") with the new parity check digit $(\underline{d}_0 \neq \underline{d}_1)$ to make a 5-digit vector. This vector is transferred to the first 5 digits of the register $\underline{o}$ which are selected by $\underline{\alpha}^5/\underline{o}$. The other 5 digits of $\underline{o}$ would be left unchanged.

In what follows, we shall require a few more

[illegible]

operators for dealing conveniently with one-dimensional arrays and a notation for two-dimensional arrays. Operations on two-dimensional arrays may be derived in a natural way from those on vectors but we shall not use them in this thesis.

Upper-case Roman letter will be used for two dimensional arrays, e.g., $\underline{M}$, and lower-case Roman letters will be used for vectors which represent registers, e.g., $\underline{d}$,$\underline{o}$. Special vectors such as $\underline{\alpha}$, $\underline{\omega}$ will be denoted by lower case Greek letters. The dimensions of vectors and matrices can be written in parentheses following the symbol but in general will be elided.

The notation for matrices is as follows

$$\underline{M}(n\text{x}m) = \begin{bmatrix} \underline{M}_0^0 & \underline{M}_1^0 & \underline{M}_2^0 & \cdots & \underline{M}_{m-1}^0 \\ \underline{M}_0^1 & \underline{M}_1^1 & \cdots & & \underline{M}_{m-1}^2 \\ \vdots & & & & \\ \underline{M}_0^{n-1} & \underline{M}_1^{n-1} & \cdots & & \underline{M}_{m-1}^{n-1} \end{bmatrix}$$

$\underline{M}^i$ denotes the ith row vector of $\underline{M}$ and $\underline{M}_j$ denotes the jth column vector of $\underline{M}$. Matrix notation will be used for groups of related registers such as the main memory of the IBM 1620. Access to a computer's memory is usually restricted to transferring an entire register

(row or column) at a time, to another register which acts as a buffer between the memory and the other registers of the computer. Hence, the transfer of information from the $i^{th}$(row) register of a memory $\underline{M}$ into a buffer $\underline{b}$ would be indicated by the statement

$$\underline{b} \leftarrow \underline{M}^i .$$

The memory of the 1620 may be described by a matrix with $10^4$ rows and 12 columns whose elements are either 0 or 1. Information may be transferred from or to the memory, twelve digits at a time.

The following operations are used extensively in describing the manipulation of information.

Rotation: The cyclical left rotation of a vector $\underline{x}$ is denoted by $k \uparrow \underline{x}$ : e.g., $3 \uparrow (1,2,3,4,5) = (4,5,1,2,3)$. The cyclical right rotation of a vector $\underline{x}$ is denoted by $k \downarrow \underline{x}$: e.g., $3 \downarrow (1,2,3,4,5) = (3,4,5,1,2)$.

Reduction: An operation $\odot$ is applied to all elements of a vector in turn to produce a scalar.
$\odot / \underline{x} \equiv (\ldots((\underline{x}_1 \odot \underline{x}_2) \odot \underline{x}_3 \ldots) \odot \underline{x}_n$
e.g., $+/(-3,4,1,2) = (((-3+4)+1)+2) = 4$
and $\neq /(1,0,1,1) = 1$ (Parity-check).
(This operation should be carefully distinguished

from the <u>selection</u> operation which involves two <u>vectors</u> e.g., $\underline{u}/\underline{x}$.)

<u>Vector Operations</u>: Operations which involve two operands (binary operations) are extended to vectors element by element. If $\odot$ is any binary operation, and $\underline{x},\underline{y}$ two vectors of the same dimension

$$\underline{z} \leftarrow \underline{x} \odot \underline{y} \text{ implies } \underline{z}_i \leftarrow \underline{x}_i \odot \underline{y}_i$$

$$i = 0,1,2,\ldots$$

<u>Scalar Multiplication</u>: $\underline{y} = k\underline{x}$ is defined by $\underline{y}_i \leftarrow k \times \underline{x}_i$ . For example, $3\,\underline{\varepsilon} = (3,3,3,\ldots,3)$ where $\underline{\varepsilon}$ is the special vector $(1,1,1,\ldots,1)$.

<u>Scalar Product</u>: If $\odot_1$ and $\odot_2$ are any two operators then it is possible to define a scalar product of two vectors $\underline{x}$ and $\underline{y}$

$$z \leftarrow \underline{x} \overset{\odot_1}{\underset{\odot_2}{}} \underline{y} (= \odot_1/(\underline{x} \odot_2 \underline{y})).$$

A special case of interest is the ordinary inner product of linear algebra. The inner product of two numerical vectors $\underline{x}$ and $\underline{y}$ is given by $z = \underline{x} \overset{+}{\underset{\times}{}} \underline{y}$. It is used frequently here to calculate the number whose binary representation

is held in a register. To illustrate, let $\underline{y}$ be the vector (1,1,0,0,1) which is a binary-coded decimal representation of the integer 9 where the leading digit, $\underline{y}_0$, is an odd parity check. Let $\underline{x} = (0,8,4,2,1)$ be a <u>weighting</u> vector whose elements are the weights associated with the corresponding elements of $\underline{y}$. Then

$$\begin{aligned}\underline{x} \overset{+}{\underset{x}{}} \underline{y} &= (0,8,4,2,1) \overset{+}{\underset{x}{}} (1,1,0,0,1)\\ &= +/((0,8,4,2,1)\times(1,1,0,0,1))\\ &= +/(0,8,0,0,1)\\ &= 9.\end{aligned}$$

<u>Residue</u>: $b|n$ is the residue of n modulo b.

e.g., $5|12 = 2$.

<u>Complement</u>: The Boolean complement of a logical variable or vector is indicated by a bar over the symbol.

e.g., $\underline{u} = (1,0,0,1,1)$

$\bar{\underline{u}} = (0,1,1,0,0)$

$\bar{\underline{\varepsilon}} = (0,0,0,\ldots)$ .

## <u>Section 5.3 Static Structure of the IBM 1620</u>

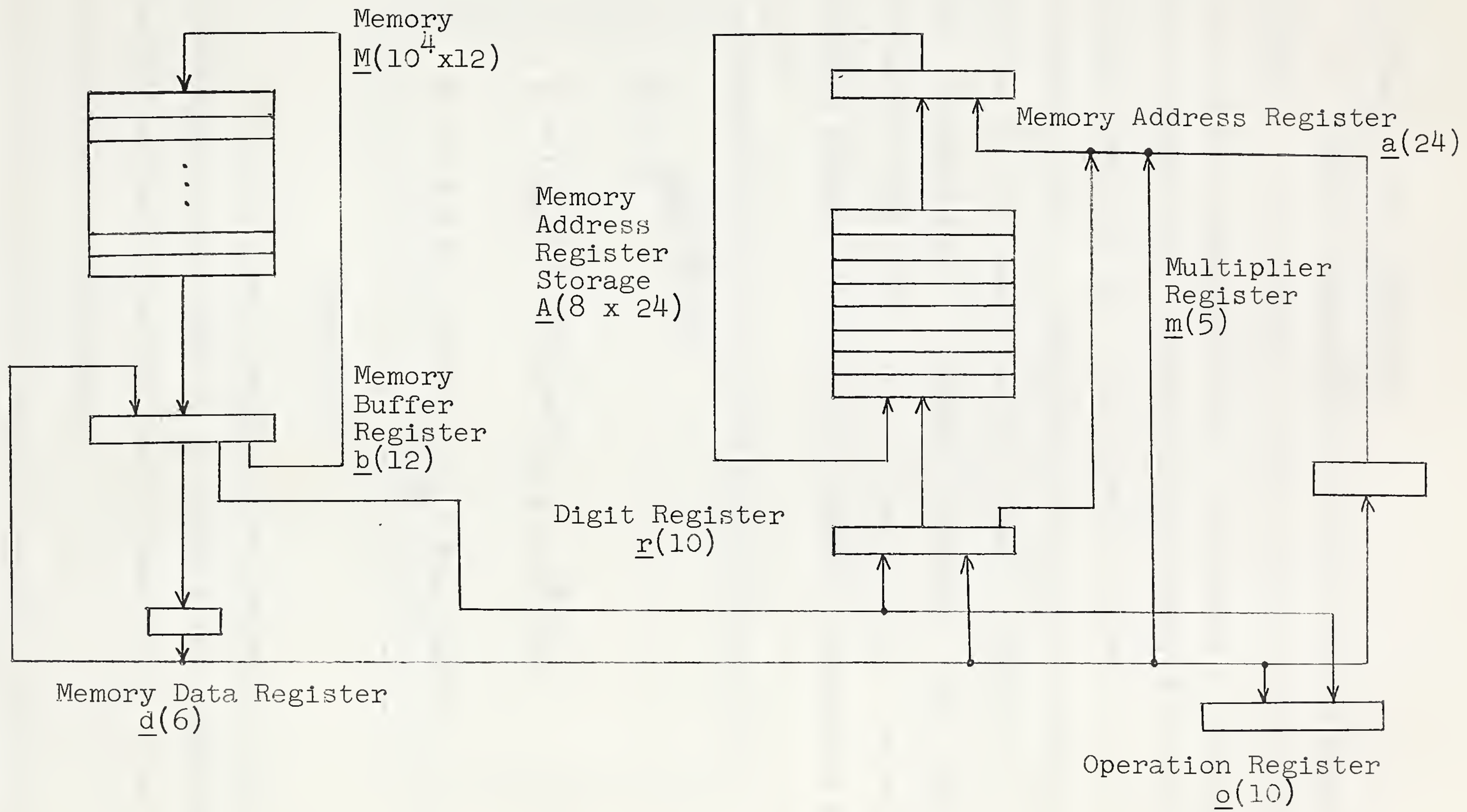
It is physically impossible to inter-connect all registers of a machine as complex as a digital computer. Hence, a transfer between two arbitrary registers must often

be accomplished by a series of transfers between intermediate registers. The permitted transfer-paths between registers and the number of digits in each register define the static structure of a machine. Fig. 5.3.1 illustrates the gross static structure of the IBM 1620. Transfer paths may be either direct transfers or transfers which include a transformation of information such as was described in Section 5.2.3.

| Name | Symbol | Dimension |
|---|---|---|
| Memory | $\underline{M}$ | $10^4 \times 12$ |
| Memory Data Register | $\underline{d}$ | 6 |
| Memory Buffer Register | $\underline{b}$ | 12 |
| Memory Address Register | $\underline{a}$ | 24 |
| Memory Address Register Storage | $\underline{A}$ | $8 \times 24$ |
| Digit Register | $\underline{r}$ | 10 |
| Operation Register | $\underline{o}$ | 10 |
| Multiplier Register | $\underline{m}$ | 5 |

Table 5.3.1

The gross functions of the machine, such as adding, multiplying, etc., which can be specified externally by a programmer are realised as sequences of transfers between the registers. A typical instruction to the machine such as adding two ten-digit decimal numbers together would require about 250 of the transfers listed in Appendix 5.a. Fortunately, these long sequences are composed of recurrent short sequences


Memory
M(10^4x12)
Memory Address Register Storage A(8 x 24)
Memory Address Register a(24)
Multiplier Register m(5)
Memory Buffer Register b(12)
Digit Register r(10)
Memory Data Register d(6)
Operation Register o(10)

STATIC STRUCTURE OF IBM 1620

FIGURE 5.3.1

of perhaps 10-20 transfers.

Before considering the very long sequences of transfers, let us examine a short sequence which occurs in all of the long sequences. The purpose of the sequence is to transfer information from the Memory $\underline{M}$ to the Operation Register $\underline{o}$. It is clear, from Fig. 5.3.1, that information must be transferred from $\underline{M}$ to the Memory Buffer Register, $\underline{b}$, and thence to the Operation Register $\underline{o}$. We find, on examining the detailed transfers, that part of the information must travel via the register $\underline{d}$. The only available sequence which will transfer information from $\underline{M}$ to $\underline{o}$ is

$$\underline{b} \leftarrow \underline{M}^{\beta \overset{+}{\times} (\bar{\underline{\omega}}^1 / \underline{a})} \qquad 1$$

$$\underline{d} \leftarrow (\bar{a}_{23}\, \underline{\epsilon} \wedge \underline{\alpha}^6 / \underline{b}) \vee (a_{23}\, \underline{\epsilon} \wedge \underline{\omega}^6 / \underline{b}) \qquad 2$$

$$\underline{o} \leftarrow (d_0 \neq d_1), (\underline{\omega}^4 / \underline{d}), (b_6 \neq b_7), (\underline{\omega}^6 / \underline{b}) \qquad 3$$

Fig. 5.3.2[+]

In the first transfer, the contents of the register $\underline{a}$ determine the row index of $\underline{M}$, since $\underline{\beta}$ is a fixed vector (defined below). Note that the last element of $\underline{a}$ is not involved in calculation of the index.

---

+ To avoid confusion between Boolean and arithmetic operations, " $\wedge$ " and " $\vee$ " will be used for "AND" and "OR" and "+" will have its usual arithmetical meaning.

The second transfer in Fig. 5.3.2 requires a choice between the first half of register $\underline{b}$, $\underline{\alpha}^6/\underline{b}$, and the second half, $\underline{\omega}^6/\underline{b}$. The choice is made on the basis of the contents of the last element of $\underline{a}$, viz. $\underline{a}_{23}$. If $\underline{a}_{23} = 0$, $\underline{\alpha}^6/\underline{b}$ is transferred to $\underline{d}$ and if $\underline{a}_{23} = 1$, $\underline{\omega}^6/\underline{b}$ is transferred to $\underline{d}$.

The third transfer trims the original 12 binary digits to fit the 10-digit $\underline{o}$ register by means of two exclusive-or operations ($\neq$). Part of this transfer was described in the example in Section 5.2.3

The sequence may be followed by means of an example. Let $\underline{a} = (1,0,0,0,1,0,0,0,0,1,0,0,1,1,0,0,1,1,1,1,0,1,0,0)$. Since $\underline{\beta} = (0,20000,10000,5000,0,4000,2000,1000,500,0,400,200,$ $100,50,0,40,20,10,5,0,4,2,1)$,

$\underline{\beta} \overset{+}{\underset{\times}{}} (\bar{\underline{\omega}}^1/\underline{a}) = +/(0,0,0,0,0,0,0,0,0,0,0,0,100,50,0,0,20,10,5,0,0,2,0)$

$= 187$.

Thus, in this example, the following transfer takes place

$$\underline{b} \leftarrow \underline{M}^{187}.$$

Suppose that $\underline{M}^{187}$ contains $(1,1,0,0,1,0,0,0,0,0,0,1)$. Then, since the last element of $\underline{a}$, ($\underline{a}_{23}$) contains 0, we may write the right-hand side of Transfer 2 as

$(1\underline{\varepsilon} \wedge \underline{\alpha}^6/\underline{b}) \vee (0\underline{\varepsilon} \wedge \underline{\omega}^6/\underline{b})$. Thus, $\underline{\alpha}^6/\underline{b}(=1,1,0,0,1,0)$

is selected for transfer to $\underline{d}$. The last transfer selects 4 digits from each of $\underline{b}$ and $\underline{d}$ and computes the digits $\underline{o}_0$ and $\underline{o}_5$ by the relations $(\underline{d}_0 \neq \underline{d}_1)$ and $(\underline{b}_6 \neq \underline{b}_7)$ so that both halves of $\underline{o}$ will have correct parity. Hence, $\underline{o}$ finally contains the vector (0,0,0,1,0,0,0,0,0,1).

The vectors which are transferred by the sequence of Fig. 5.3.2 are binary-coded decimal representations based on the 8,4,2,1 code of Chapter 2. The binary code for decimal digits used in the 1620 is given in Appendix 5.b. For example, in the $\underline{d}$ register the vector (1,1,0,0,1,0) represents the decimal digit 2. The digits $\underline{d}_2, \underline{d}_3, \underline{d}_4, \underline{d}_5$ are the 8,4,2,1 code for 2; the first two digits, $\underline{d}_0$, $\underline{d}_1$, are a parity-check digit and a "flag" digit respectively. The check digit represents an odd parity check. The flag digit is used for various purposes in the 1620 as will be seen in the next section. A flagged decimal digit is written with an overbar e.g., $\bar{2}$. Flag digits are not required in the Operation Register and, as we have seen, are removed from the binary-coded decimal representation before transfer to the Operation Register. Hence, in decimal notation, the contents of $\underline{M}^{187}$ $(\bar{2},1)$ are transferred to $\underline{o}$ as (2,1).

Each decimal digit in Memory is referred to by

a five-decimal-digit number called its address. When information is to be transferred from Memory to some other part of the machine, the binary-coded representation of the address must appear in the Memory Address Register, $\underline{a}$, at the appropriate time. In the example used above, the register $\underline{a}$ contained the binary representation of the number 374 = $(2\underline{\epsilon} \times \underline{\beta}, 1)^{+}_{\times}\ \underline{a}$. This is exactly twice the row index used to specify the vector $\underline{M}^{187}$, since each row of memory holds the binary representation of two decimal digits. The first half of each Memory register $\underline{M}^{i}$ is associated with even address 2i and the second half with odd address 2i+1. A reason for choosing this somewhat roundabout way of addressing memory is that it was probably cheaper to build a memory and its associated circuitry to operate in this fashion than in the more direct way.

It should be noted that the numerical calculation $\underline{\beta}^{+}_{\times}(\overline{\underline{\omega}}^{1}/\underline{a})$ is realized in the 1620 in terms of a fairly complex network of AND and OR gates which depends to a large extent on the physical structure of the memory of the 1620. A description of the network is not presented here as the numerical calculation $\underline{\beta}^{+}_{\times}(\overline{\underline{\omega}}^{1}/\underline{a})$ reveals more clearly the relationship between the Memory $\underline{M}$ and the Memory Address Register $\underline{a}$.

We have examined a short sequence of transfers and even that proved to be somewhat complicated. To make this and other much longer sequences understandable it is desirable to examine sequences in the context of the overall objectives they are designed to achieve. The overall objectives are, of course, the manipulation of decimal data. The tools available to the user of the machine are operations such as addition, comparison and transfer of groups of decimal digits. A logical sequence of such operations or instructions, as they are called, form the program which the machine is expected to follow. The instructions are selected from a basic set of operations built into the machine in the form of well-defined sequences of the detailed transfers listed in Appendix 5.a. This basic repertoire of instructions is the topic of the next section.

## Section 5.4 IBM 1620 Repertoire of Instructions

This section describes the decimal language the programmer uses in setting out instructions for the 1620 to follow. In reading it, it is well to keep in mind that all of these instructions are performed by means of transfers of information within the machine. This is evident in some instructions which cause explicit transfers for programming purposes but it is also true of operations such as add which is performed by looking up the sum of two digits in

[illegible]

a table stored in the Memory. (This will be discussed more fully later in the section.)

The 1620 is capable of performing about 30 different operations. Each operation and its operands is specified by a 12 decimal digit instruction consisting of 2 operation digits and two 5-digit addresses arranged as follows

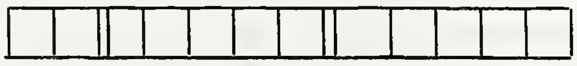


Fig. 5.4.1

Instructions are stored sequentially in Memory and are executed in this sequence. It is possible, however, to alter the sequence by means of branch instructions included in the program.

An instruction may operate on a single digit, a field, or a record. A field is a group of contiguous digits and a record is a group of contiguous fields. Since a field may consist of any number of digits greater than 1, it is necessary to specify the beginning and end of a field. The beginning of a field is specified by the address in the instruction which refers to the field; the end of the field is defined by the presence of a flag-digit in the

binary representation of a digit of the field. For example, the decimal integer 23 might be stored as a 3-digit field in memory positions 02000,02001,02002 thus

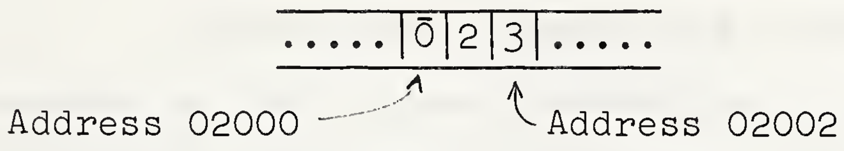


Fig. 5.4.2

(The presence of a flag in the binary representation of a decimal digit is indicated by an overbar). The beginning of the field is specified by the appearance of the address 02002 in an instruction; the end of the field is specified by the presence of the flag digit in the digit 0 in position 02000. "Beginning" and "end" imply a time sequence - the digits of the field are operated on, one at a time, in the order 3,2,$\bar{0}$.

Similarly, the digits of a record are processed in sequence. This time, the operation is terminated by a special character called a record mark (written ǂ ) and the digits are operated on in the order $\bar{0}$,2,3,...

For example, the beginning of the record might be

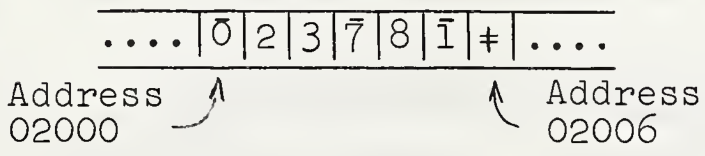


Fig. 5.4.3

specified as 02000 in an instruction and the end of the operation or the record would be determined by the presence of the record mark at address 02006.

These digits, fields and records are used as operands by the various instructions of the machine. An instruction may require no operand, one operand or two operands. A typical instruction is "transfer the field whose address is Q to the field whose address is P".

The instructions fall into four general categories:

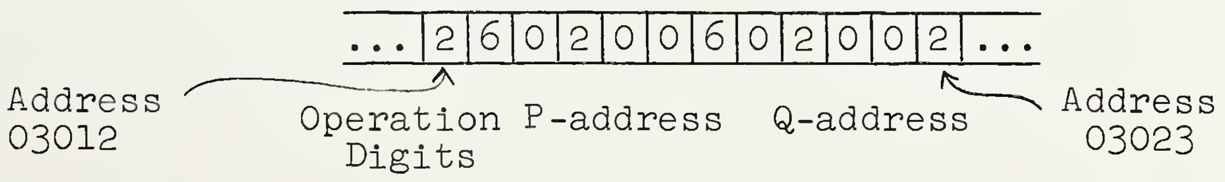
1) Data Transmission:

| Name of Instruction | Operation Digits |
|---|---|
| *Transmit Digit | 25 |
| *Transmit Field | 26 |
| Transmit Record | 31 |

Table 5.4.1

These are the "work-horse" instructions. Most of the work done by a program is expressed in terms of single transfers of digits, fields and records from place to place in Memory.

Example: The following instruction is stored in addresses



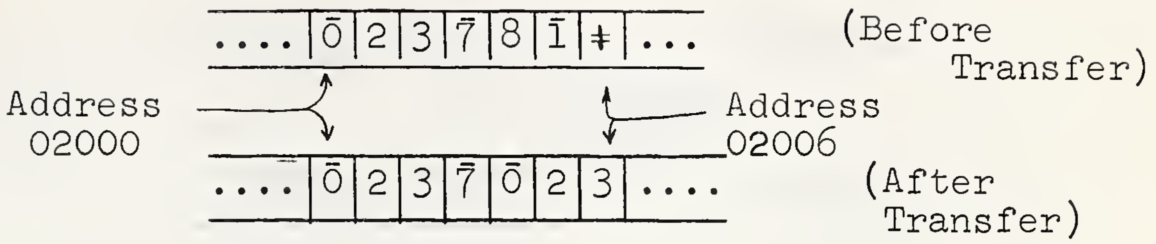


Fig. 5.4.4

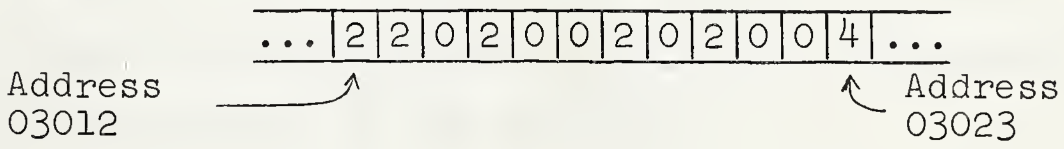
03012 through 03023. The instruction means "Transmit the field specified by the Q address (02006) to the field specified by the P-address (02002)." The contents of the relevant portion of Memory are indicated before and after the instruction is executed.

2) <u>Arithmetic Instructions</u>

| | |
|---|---|
| *Add | 21 |
| *Subtract | 22 |
| *Multiply | 23 |
| *Compare | 24 |

Table 5.4.2

<u>Example</u>



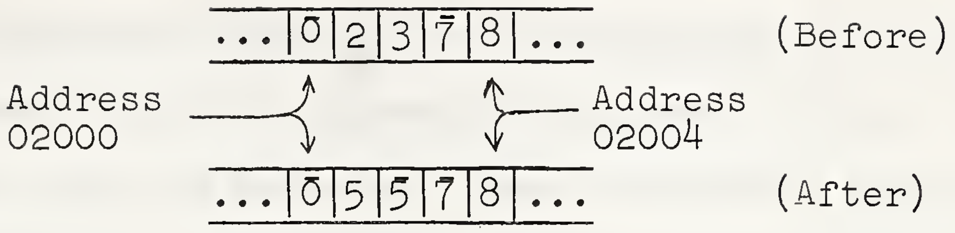

... | $\bar{0}$ | 2 | 3 | $\bar{7}$ | 8 | ...
(Before)
Address 02000
Address 02004
... | $\bar{0}$ | 5 | $\bar{5}$ | 7 | 8 | ...
(After)

Fig. 5.4.5

Arithmetic instructions operate on fields and the sign of a field is denoted by the presence (minus) or absence (plus) of a flag on the first digit of a field which is processed, i.e., the <u>right-most</u> digit as the field is written here. The instruction in Fig. 5.4.5 results in the subtraction (operation digits 22) of the field at 02004 which contains $\bar{7}8$ from the field at 02002 which contains $\bar{0}23$. The result $\bar{0}5\bar{5}$ replaces the field at 02002. If the arithmetic result would exceed the length of the P-field, the operation is terminated at the end of the P-field and a special indicator or switch is turned on (Overflow Indicator). This condition can arise if a carry is generated at the end of the P-field or if the length of the Q-field exceeds the length of the P-field. If the Q-field is shorter than the P-field, leading zeros are supplied by the machine to make up the deficit in the Q-field.

The compare operation is essentially a subtract operation which does not store the result but sets three

special indicators from which it is possible to determine whether the P-field is algebraically greater than, equal to or less than the Q-field. These indicators are called the High/Positive, Equal/Zero and High/Positive Or Equal/Zero indicators.

Arithmetic operations are performed by means of a rather unusual and interesting table-look-up scheme. Two tables are stored in Memory - the multiply table occupies the Memory area, 00100 - 00299 and the addition table occupies 00300 - 00399. The technique will be described for addition. The addition table contains the sums of all possible pairs of decimal digits. The table is so arranged that if the two digits to be added are x and y, the sum digit will be found at address 003xy. If the sum exceeds 9 a flag appears in the sum digit indicating to the machine that the next addition should include a carry. Subtraction is accomplished in the same manner except for a complementation of the digits of one field.

The interesting feature of this method of addition is that it is quite general. Since the table is stored in Memory it may be changed by programming (intentionally or unintentionally) to represent any desired binary operation. (Binary is used in the sense of involving <u>two</u> operands.)

In effect, the machine can perform directly the general operation

$$X * Y.$$

Where X and Y are elements of sets of symbols chosen from the alphabet (0,1,...,9) and * is defined by a table of X vs. Y.

3) Branch Instructions

| | |
|---|---|
| Branch | 49 |
| Branch No Flag | 44 |
| Branch No Record Mark | 45 |
| Branch On Digit | 43 |
| Branch Indicator | 46 |
| Branch No Indicator | 47 |
| *Branch and Transmit | 27 |
| Branch Back | 42 |

Table 5.4.3

Branch instructions are used to alter the sequence of execution of instructions in response to certain conditions occurring in the machine. For example, if an overflow occurred it might be necessary for the program to take a different course of action from that taken when no overflow occurred.

[illegible]

Example

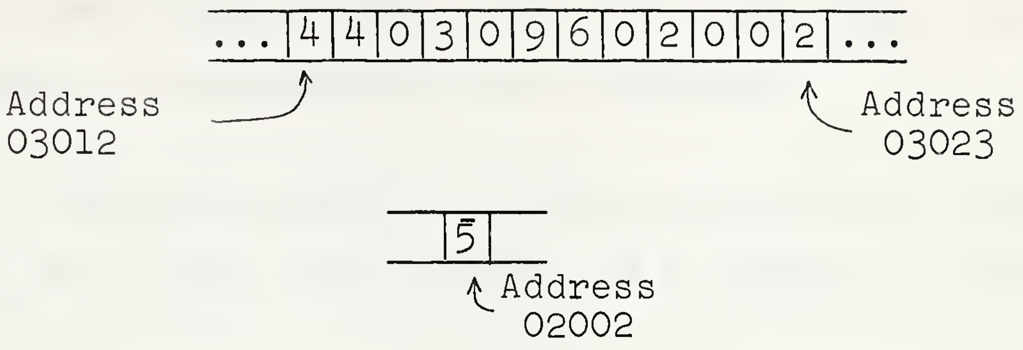


Fig. 5.4.6

The contents of the Q-Address (02002) are examined. If a flag digit is present, the next instruction in normal sequence is executed, i.e., the instruction in 03024-03035. If no flag is present, the next instruction executed is the one named by the P-address, i.e., the instruction in 03096 - 03107. Hence if the position whose address is 02002 contained $\bar{5}$, the instruction in 03024 - 03035 would be next executed.

4) Miscellaneous Instructions

| | |
|---|---|
| Set Flag | 32 |
| Clear Flag | 33 |
| Halt | 48 |
| No Operation | 41 |

Table 5.4.4

The first operation, Set Flag, is used to replace

[illegible]

an unflagged decimal digit by the corresponding flagged digit. Clear Flag performs the inverse function. Only one address is required,viz. the P-address.

The Halt operation requires no operands and its function is to halt the execution of a program at that point.

The No Operation operation,as its name suggests, performs no operation. It is occasionally useful in programming.

The instructions marked with an asterisk * have a closely related instruction form called an "immediate" instruction. In these, the Q part of the instruction is treated as a data field rather than as an address. The operation digits of an immediate instruction are formed by subtracting 10 from the corresponding normal operation digits, e.g., Add Immediate has the operation digits 11; the normal Add has the operation digits 21.

<u>Example</u>.

The function of the instruction 260200602002 in Fig. 5.4.4 could be duplicated by the immediate instruction

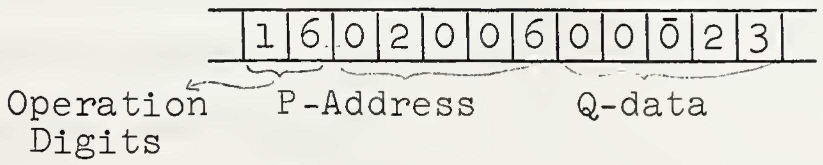


Fig. 5.4.7

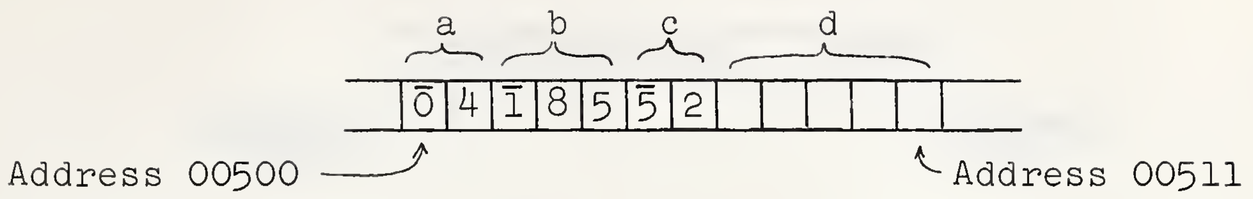
The instructions we have presented represent the basic repertoire of the 1620. (A full description of the instructions is contained in the 1620 Reference Manual (IBM [3]).) It is possible to have a few more, such as divide, built into the machine but the selection given here indicates the sort of capability one can reasonably expect of any modern computer. The 1620 differs from most other machines in two important respects. First of all, it is a variable-field-length computer and, secondly, as we have seen, the 1620 performs arithmetic by means of a table-look-up system. At the cost of complicating the logic of the machine and the programming language, the variable field length permits a wide choice of precision in arithmetic and a worthwhile economy in packing information into the Memory. In other machines, the field length is fixed at anything from 5 to 20 decimal digits (or their binary equivalents). A fixed field length usually means either wasted space in each field or, if the field length does not provide enough precision for a particular arithmetic calculation, a consequent doubling of the precision even if only a few more decimal digits are required. The choice between a variable field length and a fixed field length computer is seldom clear-cut and varies with the uses envisaged for the computer.

We conclude this section with an example of a 1620 program which will be used for illustration in the next section. The list of instructions in Fig. 5.4.8 is designed to perform the calculation d = a(b+c) where the value of a is stored as a 2-digit field at addresses 00500 - 00501 and b and c at 00502 - 00504 and 00505 - 00506 respectively. The result, d, is to be stored at addresses 00507 - 00511 either as a 4-digit field or a 5-digit field. d is to be stored as a 4-digit field if its leading digit has the value 0.

```
260051100504   1
210051100506   2
230051100501   3
430046000095   4
320009600000   5
260051100099   6
```

Fig. 5.4.8

The program (or program fragment) is stored in Memory with the first instruction in addresses 00400 - 00411 and with the other instructions immediately following. Let us examine the operation of the machine on the particular set of data in Fig. 5.4.9 as it follows the instructions.



A = +4, b = +185, c = +52

Fig. 5.4.9

Instruction 1 (Fig. 5.4.8)

The operation digits (26) are transferred from Memory to the Operation Register where they are decoded as the Transmit Field Operation . (This and other transfers must follow the paths in Fig. 5.3.1.) The two addresses (00511) and (00504) are transferred to Memory Address Register Storage. This completes the instruction interpretation phase. The machine now executes a transmit field operation under the control of the addresses 00511 and 00504 held in Memory Address Register Storage. The digit at 00504 is transmitted to the Memory Buffer Register and from there back into the Memory to the address 00511. Similarly the digit at 00503 is transmitted to 00510. Finally, the digit at 00502 (which contains a field flag) is transmitted to 00509. This terminates the execution of Instruction 1. The contents of the appropriate section of memory is indicated in Fig. 5.4.10.

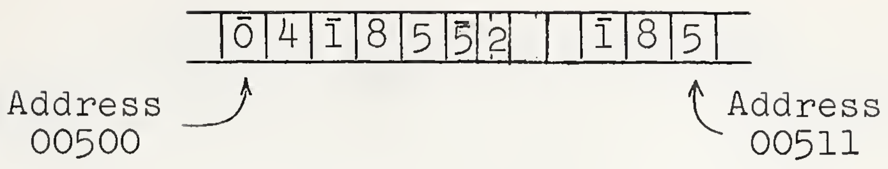


Fig. 5.4.10

Instruction 2.

The operation digits (21) are transferred to the Operation Register and decoded as the Add Operation. The two addresses 00511 and 00506 are entered into Memory Address Register Storage. The addition now proceeds digit by digit. The sum of the digit in 00511(5) and the digit in 00506(2) is looked up in the add table stored in Memory at 00300 - 00399. The sum digit (7) is placed in 00511. Next the digit in 00510(8) is added to the digit in 00505($\bar{5}$). First the flag on the $\bar{5}$ is removed and noted as a field flag so that leading zeros will be supplied as needed in later additions. Then the sum of 8 and 5 is looked up in the table. This is found to be $\bar{3}$. The flag on this sum digit indicates that a carry should occur in the next addition. The sum digit 3 is stored in Memory at 00510. Finally, the leading zero supplied by the machine is added to the carry from the last addition to give the partial sum of 1. The sum of this digit and the 1 from address 00509 is looked up in the table and the result placed in 00509. The operation terminates here because of the field flag in 00509. The results are shown in Fig. 5.4.11.

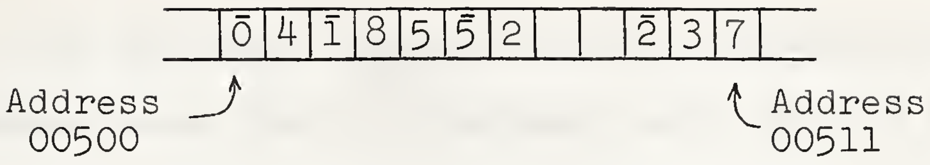
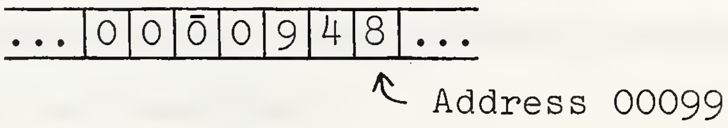


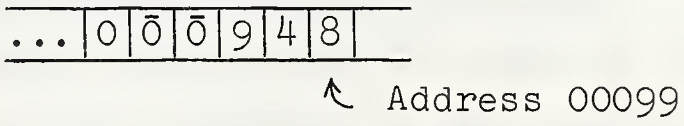
Fig. 5.4.11

Continuing in this fashion the machine achieves the objectives laid out in the program. The results of the remaining instructions are shown in Fig. 5.4.12

Instruction 3.



Instruction 5.



Instruction 6.

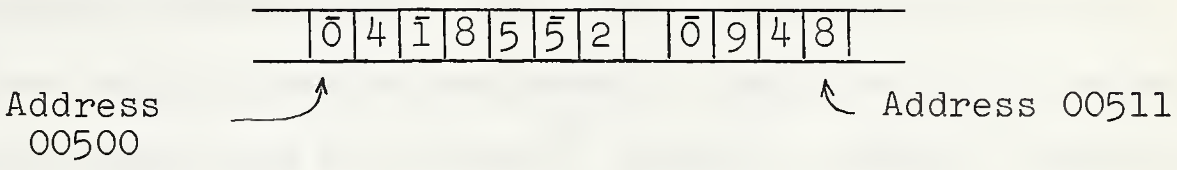


Fig. 5.4.12

Instruction 4 makes the decision to shorten the field to four digits by placing a flag on the digit in address 00095 since the digit in 00095 has the value 0.

It is clear from the detailed description of these instructions that a programmer must present specify very precisely what he wants the machine to do. Some of the apparent complication arises because the machine is performing in a mechanical way operations which human beings have learned to do in an intuitive manner. In particular, the machine depends heavily on the presence of flags on the data to control the operations, i.e., it demands explicit indications of field lengths etc., to which it can respond in a deterministic way. The mechanism of the machine's response will become clear in the next section.

## Section 5.5 Dynamic Structure of the IBM 1620

The dynamic structure of the 1620 is defined as the set of all permitted sequences of the detailed transfers of Section 5.3. These sequences are generated by all of the possible instructions which can be formed out of two operation digits and two addresses. There are 26 (or more) possible operations and $(20000)^2$ possible address combinations. Further, the data contained in the Memory has an influence on the sequences as well. Clearly, we are dealing with an extremely large number of sequences some of which are infinite in length. Fortunately, it is possible to display the structure of the machine in the form of a directed graph (Ore) or state diagram (Bartee). Such a diagram evolves

quite naturally from considering at a gross level the successive states the machine assumes during the execution of a program and working down to the detailed transfer level by including more information in each diagram.

## Section 5.5.1 State Diagrams

The first level is very simple. The machine is either interpreting an instruction or executing an instruction. This may be indicated by the directed graph in Fig. 5.5.1 where I indicates the interpretation state and E the execution state.

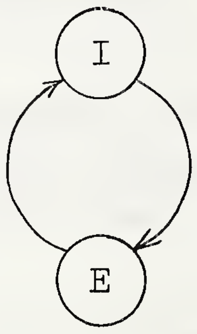


Fig. 5.5.1

There is only one possible sequence viz. I,E,I,E,... .

The second level follows from an examination of the 1620 instruction set. If we group the instructions on the basis of similarities in their execution phases, the instructions fall into eight categories (Table 5.5.1).

Table 5.5.1

| Category | Operations |
|---|---|
| $E_1$ | Transmit Digit (15,25), Transmit Field (16,26), Transmit Record (31) |
| $E_2$ | Add (11,21), Subtract (12,22), Compare (14,24) |
| $E_3$ | Multiply (13,23) |
| $E_4$ | Branch and Transmit (17,27) |
| $E_5$ | Branch Back (42) |
| $E_6$ | Set Flag (32), Clear Flag (33), Branch On Digit (43), Branch No Flag (44), Branch No Record Mark (45). |
| $E_7$ | Halt (48), No Operation (41). |
| $E_8$ | Branch Indicator (46), Branch No Indicator (47), Branch (49). |

For example, the Transmit Digit (15,25), Transmit Field (16,26) and Transmit Record (31) instructions each involve digit-by-digit transfers of information from one part of memory to another. The Add (11,21) Subtract (12,22) and Compare (14,24) are almost identical except for

the manipulation of the signs of fields. The compare operation is a Subtract without the storing of a result. The Multiply (13,23) is in a class by itself although it is performed by repeated additions and might possibly have been able to make use of the existing addition state. Branch Back is a peculiar instruction and bears no resemblance to any other, so it also is in a class by itself. The Branch and Transmit instructions (17,27) actually lie in two categories ($E_4$ and $E_1$). The branch part is executed in $E_4$ and the instruction then acts like a Transmit Field operation (16,26). The other multiple group consists of the Set and Clear Flag operations (32,33) and Branch On Digit (43), Branch No Flag (44) and Branch No Record Mark (45). These are related to each other in that they involve a transfer of a single digit from Memory for examination or modification and a transfer back of a digit to the <u>same</u> place. The Set Flag and Clear Flag Operations are completed by returning to the I-state while the Branch instructions require a further operation which is identical with Branch (49). A similar consideration places the Branch Indicator (46) and Branch No Indicator (47) with Branch (49) in the category $E_8$. The final group is $E_7$ consisting of No Operation (41) and Halt (48). They are similar in that they require no execution state. The state

diagram corresponding to Table 5.5.1 is shown in Fig. 5.5.2.

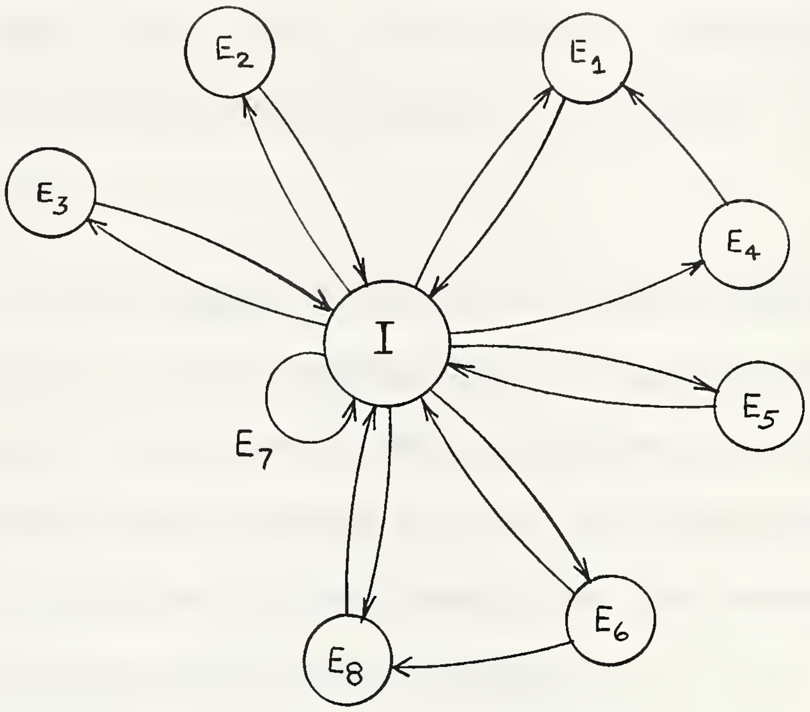


Fig. 5.5.2

The considerations which led to the state diagram of Fig. 5.5.2 correspond to the first steps the structural designer might take in an attempt to organise the instructions and, perhaps, economize on the number of states. Other combinations would be tried of course; the one given here corresponds to the current structure of the 1620.

Fig. 5.5.2 displays all the possible sequences of states the machine assumes as it follows a program. In this case the paths are defined by the information in the

[illegible]

[illegible]

[illegible]

[illegible]

operation digits of a sequence of instructions. For example, the program of Fig. 5.4.8 whose operation digits were in order, 26 , 21 , 23 , 43 , 32 , 26 would force the machine through the sequence of states $I,E_1,I,E_2,I,E_3,I,E_6,E_8,I,E_6,I,E_1$.

This diagram is still a rather gross description of the states of the machine and it may be usefully elaborated by considering the responses the machine must make to conditions arising out of the manipulation of information it picks up from Memory in the course of interpreting and executing instructions.

Let us examine the interpretation state first. An instruction is interpreted by placing the operation digits in the operation register $\underline{o}$, and placing the two addresses in the instruction in Memory Address Register Storage $\underline{A}$, at $\underline{A}^3$ and $\underline{A}^4$. If the instruction is an immediate instruction,the address of the last digit of the instruction is placed in Memory Address Register Storage rather than the data the instruction contains. This would indicate that there are four main stages in the interpretation state. However, the fundamental transfers in the machine are the transfers of two decimal digits to or from Memory and they occur in at fairly regular intervals

in both the interpretation and execution phases. If we organise the states of the machine around these fundamental transfers the structure of the machine falls into a satisfactory pattern.

On the basis of this rather loose definition of a state, the interpretation phase is seen to consist of 8 states; State 1 transfers the operation digits to the operation register, states 2,3,4 set up the first address in Memory Address Register Storage, states 5,6,7 set up the second address, and state 8 makes the proper adjustment for immediate instructions. State 8 also makes the decision, on the basis of the contents of the operation register, which execution state is to follow. Hence, the I-state may be elaborated to

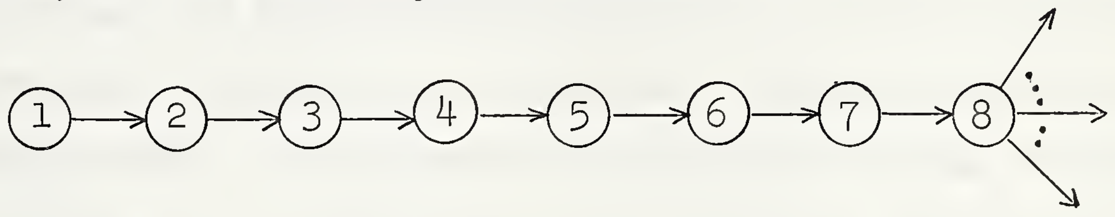


Fig. 5.5.3

In the same way, we may take each E-state and elaborate it on the basis of our new definition of a state. For example, in $E_1$, there are two fundamental transfers, one to select a digit from memory and the other to place it

back in memory in a different place. This is done once for Transmit Digit (15,25) and repeated for Transmit Field (16,26) and Transmit Record (31) until a field flag and a record mark, respectively, are detected. Then the machine returns to the beginning of the interpretation phase. This results in the diagram of Fig. 5.5.4

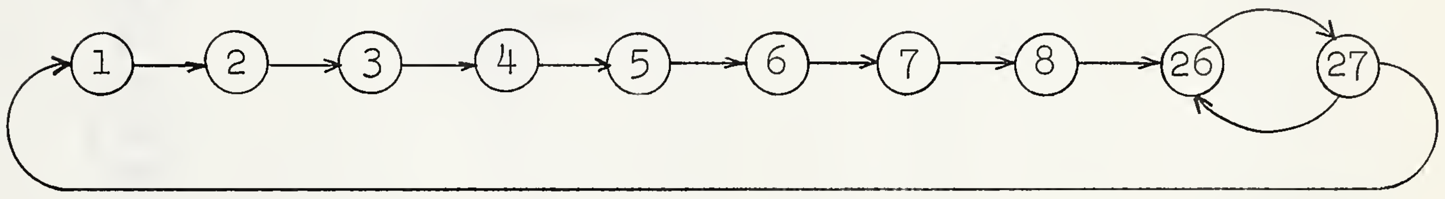
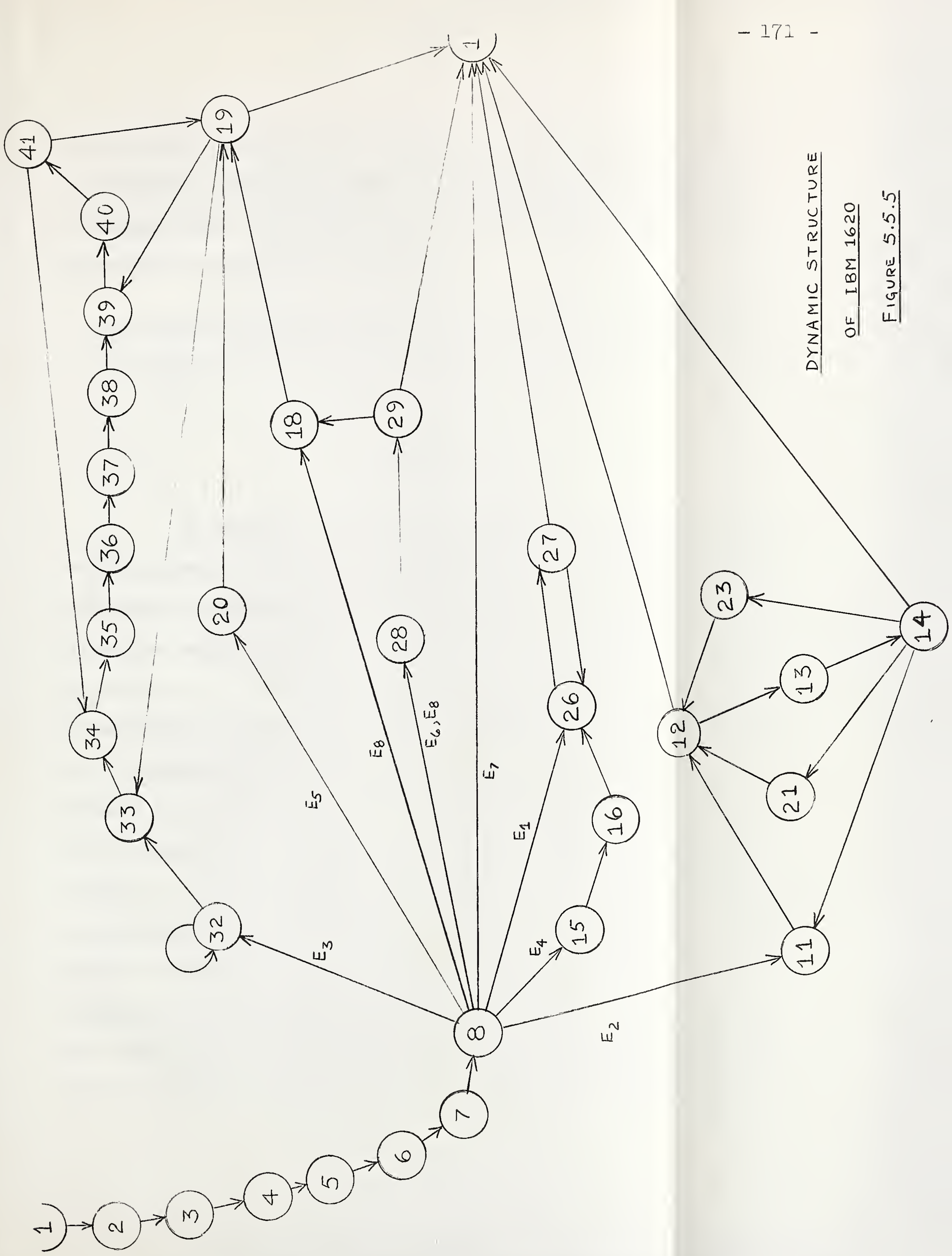


Fig. 5.5.4

If this procedure is repeated for each of the remaining E-states the diagram of Fig. 5.5.5 may be constructed. Conceivably, the procedure could be continued for finer and finer differentiations of states but this particular diagram is a compromise between too little detail and too much. In any event, each state of Fig. 5.5.5 may be related to a small set of the detailed transfers of Section 5.3 so that it is a reasonably easy step from the diagram to the most detailed transfer level. For example, when the machine is in state 1, it executes the set of transfers in Fig. 5.3.2. The diagram is useful in visualizing the dynamic structure of the machine and may be used in much the same way as Fig. 5.3.1 (another directed graph) is used in visualizing the static structure. To

[illegible]



DYNAMIC STRUCTURE

OF IBM 1620

FIGURE 5.5.5

illustrate the use of the diagram, a Transmit Field (26) instruction operating on a 3-digit field and followed by a Branch (49) instruction would cause the machine to pass through the succession of states:

$$\underbrace{1,2,3,4,5,6,7,8}_{I},\underbrace{26,27,26,27,26,27}_{E_1},\underbrace{1,2,3,4,5,6,7,8}_{I},\underbrace{18,19}_{E_8}.$$

Fig. 5.5.6

In itself, the diagram suggests a number of rather subtle ideas. For example, apart from the major loop involving the interpretation phase 1-8, it is clearly possible that the machine may be "stuck in a loop" forever, e.g., 26, 27,26,27,... . This is indeed what can happen as many 1620 programmers are painfully aware. Moreover, it suggests that the interpretation phase is on a par with the execution phases as indeed it is, since the interpretation is carried out by means of the same sort of transfers as the execution phases. As a final point, it might be noted that the (arbitrary) numbers associated with the states may be encoded as 6-binary-digit vectors and the sequence of states defined by the diagram could be generated as a sequence of vectors in much the same way as a sequence of binary vectors was generated by the encoders and decoders of Chapter 4. However, this

takes us into the realm of speculation and we must now consider the relationship of the diagram to its physical realization in the machine.

Let us associate a binary storage element with each state in the diagram. If the storage element corresponding to a particular state has the value 1, then that state may be said to be active or in control of the machine. In other words, when a control element has the value 1, the set of detailed transfers associated with it will be executed. These storage elements can clearly be regarded as elements of a register and transfer of information to and from this register may be carried out in exactly the same manner as for any of the static registers of the machine. Further, the information associated with the decision as to which state is next, may be organised into registers. Collectively, these registers are referred to as "control registers" to distinguish them from the static registers. The permitted set of transfers between the control registers and the static registers define the machine completely.

## Section 5.5.2 Control Registers

The numbering of the states in Fig. 5.5.5 is based

on the numbering used in the 1620 Manual of Instruction (IBM [2]) to denote the "timing triggers" which are the physical realizations of the states of the machine. Each state corresponds to an element of a vector $\underline{t}$ whose elements may have the value 0 or 1. When the element $\underline{t}_i$ has the value 1, the machine is defined to be in State i. Only one element may have the value 1 at any given instant of time, i.e., $+/(\underline{t}) = 1$. (It will be noted that not all of the elements of $\underline{t}$ are used in Fig. 5.5.5, e.g., $\underline{t}_9$, $\underline{t}_{10}$ etc. These correspond to triggers whose functions are not important here.) The $\underline{t}$ vector is the primary control register of the machine which passes from state to state as selected by the position of the single 1 in $\underline{t}$.

During the execution of an instruction the machine must respond to certain events occurring while the data is being manipulated, e.g., field flags. The occurrence of these events is signalled by the presence of a 1 in certain storage elements in the machine. Again it is convenient to combine these in a register (for notational convenience only). Table 5.5.2 lists the elements of the register together with the names used in IBM [2]. Some of these are self-explanatory; the others will be explained later.

| Element | Name of element |
|---|---|
| $\underline{s}_0$ | Digit |
| $\underline{s}_1$ | True/Complement |
| $\underline{s}_2$ | Recomplement |
| $\underline{s}_3$ | Recomplement Control |
| $\underline{s}_4$ | Field Mark No. 1 |
| $\underline{s}_5$ | Field Mark No. 2 |
| $\underline{s}_6$ | Record Mark |
| $\underline{s}_7$ | Carry In |
| $\underline{s}_8$ | Carry Out |
| $\underline{s}_9$ | Branch Test |
| $\underline{s}_{10}$ | First Cycle |
| $\underline{s}_{11}$ | Cycle Control |
| $\underline{s}_{12}$ | Clear Flag |
| $\underline{s}_{13}$ | Set Flag |

Table 5.5.2

$\underline{s}_1$ and $\underline{s}_7$ control the complementation of digits in arithmetic operations. $\underline{s}_{12}$ and $\underline{s}_{13}$ are used to control the setting or clearing of flags on digits to be placed in Memory.

The third control register ($\underline{c}$) is connected with the decoding of the operation digits of an instruction during the interpretation phase. When the operation digits are read into $\underline{o}$ (by $\underline{t}_1$) the following transfers take place:

$$\underline{c} \leftarrow \bar{\underline{\varepsilon}}$$

$$\underline{c}_{(\underline{\delta} \overset{+}{\underset{\times}{}} \underline{o})} \leftarrow 1$$

Fig. 5.5.7

For example, if the operation digits are 21 then a 1 is transferred to $\underline{c}_{21}$. The presence of a 1 in $\underline{c}_k$ conveys the information that operation k is to be performed. This information is mainly used to direct the machine to select the correct path during state $\underline{t}_8$ (Fig. 5.5.5)

The fourth and last control register ($\underline{i}$) is similar to the register $\underline{s}$ except that the information in $\underline{i}$ is available to the programmer by means of the Branch Indicator (46) and Branch No Indicator (47) instructions. These instructions have the form

```
46 PPPPP   0II00
47 PPPPP   0II00
```

Fig. 5.5.8

where PPPPP is the address to which a branch occurs if the condition implied by the instruction is satisfied and II are two decimal digits which define the particular indicator.

During the interpretation of the instruction, the digits II will appear in the Digit Register $\underline{r}$ during $\underline{t}_6$, when the following transfer occurs:

$$\boxed{\underline{s}_9 \leftarrow \underline{i}_{(\underline{\delta} \overset{+}{\times} \underline{r})} \wedge (\underline{c}_{46} \vee \underline{c}_{47})}$$

Fig. 5.5.9

The $\underline{i}$ register contains information about conditions occurring in the machine and about the results of previous instructions. Hence, if $\underline{i}_{12}$ had the value 1, indicating a zero result in the last arithmetic operation, then the Branch Test Switch, $\underline{s}_9$ would be set to the value 1. The Branch Test switch is interrogated during $\underline{t}_8$ to decide whether to proceed in sequence or cause a branch to address PPPPP. Table 5.5.3 list the indicators of interest here.

| Element of $\underline{i}$ | Indicator |
|---|---|
| $\underline{i}_1$ | Program Switch No. 1 |
| $\underline{i}_2$ | Program Switch No. 2 |
| $\underline{i}_3$ | Program Switch No. 3 |
| $\underline{i}_4$ | Program Switch No. 4 |
| $\underline{i}_{11}$ | High/Positive |
| $\underline{i}_{12}$ | Equal/Zero |
| $\underline{i}_{13}$ | High/Positive Or Equal/Zero ($\underline{i}_{13} = \underline{i}_{11} \vee \underline{i}_{12}$) |

(table continued on next page)

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

| Element of $\underline{i}$ | Indicator |
|---|---|
| $\underline{i}_{14}$ | Overflow |
| $\underline{i}_{16}$ | Parity Error in $\underline{\alpha}^6/\underline{b}$ |
| $\underline{i}_{17}$ | Parity Error in $\underline{\omega}^6/\underline{b}$ |

Table 5.5.3

(Program Switches are mechanical switches the operator of the machine may use to exert control over a program while it is being executed, if it contains the appropriate Branch Indicator instructions.)

To summarize, the machine is controlled by the contents of four registers $\underline{t}$,$\underline{s}$,$\underline{c}$,$\underline{i}$, of which the first is by far the most important. Each element of $\underline{t}$ is associated with a set of transfers. These transfers involve both the static registers and the control registers of the machine. <u>In particular, each element of t specifies another element of $\underline{t}$ uniquely either unconditionally or conditionally</u>. Hence, the dynamic structure illustrated in Fig. 5.5.5 is built into the total set of transfers available to the machine. The states in the diagram with one exit arrow correspond to the elements of $\underline{t}$ which specify a next $\underline{t}$ element unconditionally and the states with two or more exits correspond to $\underline{t}$ elements which specify a choice between possible next $\underline{t}$ elements.

The conditions which determine the next $\underline{t}$ element are contained in the other three control registers. For example, the choice of which path to take after $\underline{t}_8$, is made by means of the transfers:

$$\boxed{\begin{array}{l}
\underline{t}_{26} \leftarrow \underline{c}_{25} \vee \underline{c}_{26} \vee \underline{c}_{31} \\
\underline{t}_{11} \leftarrow \underline{c}_{21} \vee \underline{c}_{22} \vee \underline{c}_{24} \\
\underline{t}_{32} \leftarrow \underline{c}_{23} \\
\underline{t}_{15} \leftarrow \underline{c}_{27} \\
\underline{t}_{20} \leftarrow \underline{c}_{42} \\
\underline{t}_{28} \leftarrow \underline{c}_{32} \vee \underline{c}_{33} \vee \underline{c}_{43} \vee \underline{c}_{44} \vee \underline{c}_{45} \\
\underline{t}_{18} \leftarrow (\underline{c}_{46} \wedge \underline{s}_9) \vee (\underline{c}_{47} \wedge \bar{\underline{s}}_9) \vee \underline{c}_{49} \\
\underline{t}_{1} \leftarrow (\underline{c}_{46} \wedge \bar{\underline{s}}_9) \vee (\underline{c}_{47} \wedge \underline{s}_9) \vee \underline{c}_{41} \\
\underline{t}_{8} \leftarrow 0
\end{array}}$$

Fig. 5.5.10

Placing these transfers in sequence appears to violate the rule that only one 1 should appear in the $\underline{t}$ register at a time. In their physical realisation these transfers would occur virtually simultaneously. This is a problem for the circuit designer and we will assume that a related group of transfers to $\underline{t}$, while written sequentially,

will occur simultaneously. In any case, the 1620 has been known to attempt to perform the transfers related to two different t elements at the same time in executing certain illegal instructions.

We are now in a position to follow the sequence of transfers the machine will perform to carry out the execution of a program, if we have a description of all the possible transfers associated with each element of t. Such a listing of all of the transfers is not given here because of its length. Our purposes will be served by the detailed example given in the next section.

## Section 5.6 Detailed Interpretation and Execution

In this section we shall present the detailed sequence of transfers required for the interpretation and execution of the instruction in Fig. 5.6.1, operating on the

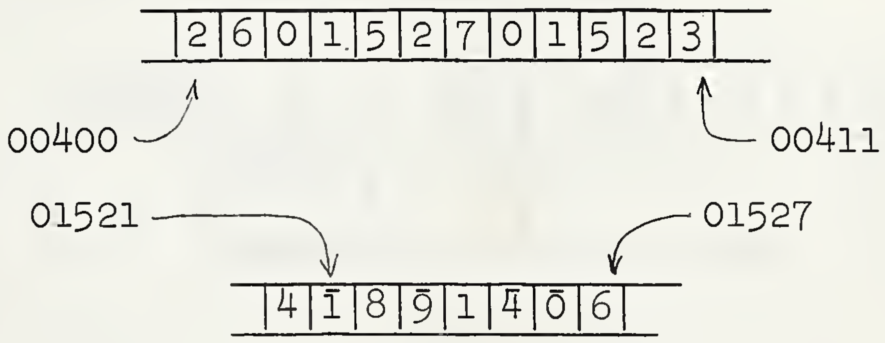


Fig. 5.6.1

data indicated. After the execution of the instruction the contents of memory will appear as in Fig. 5.6.2

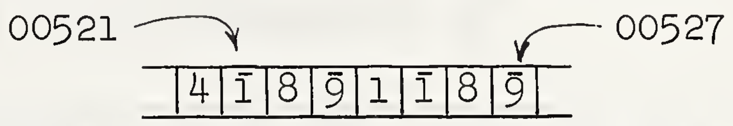


Fig. 5.6.2

In interpreting and executing this instruction the machine will pass through the states 1,2,3,4,5,6,7,8, 26,27,26,27,26,27. We shall first follow the functions the states perform in decimal notation and then examine the corresponding detailed transfers. In what follows, the general function of each state is given with the initial and final contents of the relevant registers. $\underline{A}^0$, the address of the instruction, is set to 00400.

<u>State 1</u> Copy the digits whose address is given by $\underline{A}^0$ into the operation register ($\underline{o}$) for decoding. Set the $\underline{s}$ register to zero. Add 2 to the contents of $\underline{A}^0$

| | STATE | $\underline{A}^0$ | $\underline{A}^2$ | $\underline{A}^3$ | $\underline{o}$ | $\underline{d}$ | $\underline{s}_{10}$ | $\underline{s}_4$ |
|---|---|---|---|---|---|---|---|---|
| (Initial) | 1 | 00400 | - | - | - | - | - | - |
| (Final) | | 00402 | - | - | 26 | - | 0 | 0 |

<u>State 2</u> Copy the first two digits whose address is given by $\underline{A}^0$ from Memory and place them in the first two digits of $\underline{A}^3$. Add 2 to the contents of $\underline{A}^0$

| | STATE | $\underline{A}^0$ | $\underline{A}^2$ | $\underline{A}^3$ | $\underline{o}$ | $\underline{d}$ | $\underline{s}_{10}$ | $\underline{s}_4$ |
|---|---|---|---|---|---|---|---|---|
| (Initial) | 2 | 00402 | - | - | 26 | - | 0 | 0 |
| (Final) | | 00404 | - | 01- | 26 | - | 0 | 0 |

<u>State 3</u> Copy the two digits whose address is given by $\underline{A}^0$ into the second two digits of $\underline{A}^3$. Add 2 to the contents of $\underline{A}^0$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 00404 | - | 01- | 26 | - | 0 | 0 |
| | 00406 | - | 0152- | 26 | - | 0 | 0 |

<u>State 4</u> Copy the digit whose address is given by $\underline{A}^0$ into the fifth position of $\underline{A}^3$. (This completes the storage of the P-address of the instruction in $\underline{A}^3$.) Add 1 to the contents of $\underline{A}^0$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 00406 | - | 0152- | 26 | - | 0 | 0 |
| | 00407 | - | 01527 | 26 | - | 0 | 0 |

<u>State 5</u> Copy the digit whose address is given by $\underline{A}^0$into the first position of $\underline{A}^2$. Add 2 to the contents of $\underline{A}^0$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 00407 | - | 01527 | 26 | - | 0 | 0 |
| | 00409 | 0- | 01527 | 26 | - | 0 | 0 |

<u>State 6</u> Copy the digits whose address is given by $\underline{A}^0$ into the next two positions of $\underline{A}^2$. Add 2 to the contents of $\underline{A}^0$.

| | STATE | $\underline{A}^0$ | $\underline{A}^2$ | $\underline{A}^3$ | $\underline{o}$ | $\underline{d}$ | $\underline{s}_{10}$ | $\underline{s}_4$ |
|---|---|---|---|---|---|---|---|---|
| (Initial) | 6 | 00409 | 0- | 01527 | 26 | - | 0 | 0 |
| (Final) | | 00411 | 015- | 01527 | 26 | - | 0 | 0 |

<u>State 7</u> Copy the digits whose address is given by $\underline{A}^0$ into the last two positions of $\underline{A}^2$. Add 1 to the contents of $\underline{A}^0$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 00411 | 015- | 01527 | 26 | - | 0 | 0 |
| | 00412 | 01523 | 01527 | 26 | - | 0 | 0 |

<u>State 8</u> In this example, this state merely sets the First Cycle switch on, i.e., $\underline{s}_{10} = 1$ and sends the machine on to state 26 since the operation was decoded as a Transmit Field (26) instruction. (The numbering of the operation and the state is coincidental only.)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 8 | 00412 | 01523 | 01527 | 26 | - | 0 | 0 |
| | 00412 | 01523 | 01527 | 26 | - | 1 | 0 |

<u>State 26</u> Copy the digit whose address is given by $\underline{A}^2$ into the Memory Data Register $\underline{d}$. Subtract 1 from the contents of $\underline{A}^2$. (First cycle.)

| | STATE | $\underline{A}^0$ | $\underline{A}^2$ | $\underline{A}^3$ | $\underline{o}$ | $\underline{d}$ | $\underline{s}_{10}$ | $\underline{s}_4$ |
|---|---|---|---|---|---|---|---|---|
| (Initial) | 26 | 00412 | 01523 | 01527 | 26 | - | 1 | 0 |
| (Final) | | 00412 | 01522 | 01527 | 26 | $\bar{9}$ | 1 | 0 |

<u>State 27</u> Copy the digit in $\underline{d}$ into Memory at the address given by $\underline{A}^3$. Subtract 1 from the contents of $\underline{A}^3$. Turn OFF First Cycle Switch, i.e., $\underline{s}_{10} \leftarrow 0$. Go to State 26 if $\underline{s}_4 = 0$. Go to state 1 if $\underline{s}_4 = 1$. (First cycle.)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 00412 | 01522 | 01527 | 26 | $\bar{9}$ | 1 | 0 |
| | 00412 | 01522 | 01526 | 26 | $\bar{9}$ | 0 | 0 |

<u>State 26</u> Copy the digit whose address is given by $\underline{A}^2$ into $\underline{d}$. Subtract 1 from the contents of $\underline{A}^2$. If $\underline{s}_{10} = 0$, examine the digit in $\underline{d}$. If it has a flag, set $\underline{s}_4 = 1$. ($\underline{s}_4$ is the Field Mark No. 1 switch which indicates that a <u>field</u> flag rather than a <u>sign</u> flag (First Cycle) has been detected.)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 26 | 00412 | 01522 | 01526 | 26 | $\bar{9}$ | 0 | 0 |
| | 00412 | 01521 | 01526 | 26 | 8 | 0 | 0 |

<u>State 27</u> Copy the digit in $\underline{d}$ into Memory at the address given by $\underline{A}^3$. Subtract 1 from the contents of $\underline{A}^3$. Go to state 26 if $\underline{s}_4 = 0$.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 00412 | 01521 | 01526 | 26 | 8 | 0 | 0 |
| | 00412 | 01521 | 01525 | 26 | 8 | 0 | 0 |

<u>State 26</u> Copy the digit whose address is given by $\underline{A}^2$ into $\underline{d}$. Subtract 1 from the contents of $\underline{A}^2$. If $\underline{s}_{10}=0$, examine the contents of $\underline{d}$. If it has a flag, set $\underline{s}_4=1$.

| | STATE | $\underline{A}^0$ | $\underline{A}^2$ | $\underline{A}^3$ | $\underline{o}$ | $\underline{d}$ | $\underline{s}_{10}$ | $\underline{s}_4$ |
|---|---|---|---|---|---|---|---|---|
| (Initial) | 26 | 00412 | 01521 | 01525 | 26 | 8 | 0 | 0 |
| (Final) | | 00412 | 01520 | 01525 | 26 | $\bar{1}$ | 0 | 1 |

<u>State 27</u> Copy the digit in $\underline{d}$ into Memory at the address given by $\underline{A}^3$. Subtract 1 from the contents of $\underline{A}^3$. Go to state 26 if $\underline{s}_4$ = 0. Go to state 1 if $\underline{s}_4$ = 1.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 00412 | 01520 | 01525 | 26 | $\bar{1}$ | 0 | 1 |
| | 00412 | 01520 | 01524 | 26 | $\bar{1}$ | 0 | 1 |

Since $\underline{s}_4$ = 1 this time, the machine returns to state 1 where it will interpret the instruction whose address is contained in $\underline{A}^0$(00412).

With this decimal version of the sequence of states as a guide, we can examine the transfers which the machine actually uses to interpret and execute the instruction. We shall list the set of transfers for each state and follow it with a discussion. Since these are the general transfers for the states, they represent the interpretation of all instructions in the 1620's repertoire and the execution of all of the operations in the category $E_1$ of Table 5.5.1. In the discussion following each state, the general verbal statement of the function of each transfer followed in square brackets by the results for the particular case we have just considered.

It should be kept in mind throughout that these transfers deal with <u>binary</u> vectors although for brevity their decimal equivalents will be used. The word "digit" used by itself should be taken to mean "binary digit."

We start with $\underline{t}_1 = 1$ and the address of the first digit of the instruction in $\underline{A}^0$, i.e., $\underline{A}^0 = \underline{bcd}(00400)$. The The transfers connected with $\underline{t}_1$ are:

$$\boxed{\begin{array}{rl}
1 & \underline{a} \leftarrow \underline{A}^0 \\
2 & \underline{b} \leftarrow \underline{M}^{\underline{\beta}} \overset{+}{\times} (\underline{\bar{\omega}}^{1}/\underline{a}) \\
3 & \underline{d} \leftarrow (\underline{\bar{a}}_{23}\underline{\epsilon} \wedge \underline{\alpha}^{6}/\underline{b}) \vee (\underline{a}_{23}\underline{\epsilon} \wedge \omega^{6}/\underline{b}) \\
4 & \underline{o} \leftarrow (\underline{d}_0 \neq \underline{d}_1), (\underline{\omega}^{4}/\underline{d}), (\underline{b}_6 \neq \underline{b}_7), (\underline{\omega}^{4}/\underline{b}) \\
5 & \underline{c} \leftarrow \underline{\bar{\epsilon}} \\
6 & \underline{c}^{(\underline{\delta} \underset{\times}{+} \underline{o})} \leftarrow 1 \\
7 & \underline{A}^0 \leftarrow \underline{bcd}\ (20000 \mid (\underline{\gamma} \underset{\times}{+} \underline{a} + 2)) \\
8 & \underline{s} \leftarrow \underline{\bar{\epsilon}} \\
9 & \underline{s}_1 \leftarrow 1 \\
10 & \underline{t}_2 \leftrightarrow \underline{t}_1
\end{array}}$$

Fig. 5.6.3

1. Transfer the address of instruction to Memory Address Register $\underline{a}$. [00400]

2. Select 12 digits from $\underline{M}$. [$\underline{M}^{200} = (0,0,0,0,1,0,1,0,0,1,1,0) \Rightarrow 26$].

3. Transfer the first half of $\underline{b}$ to $\underline{d}$ if $\underline{a}_{23} = 0$ or the second half of $\underline{b}$ to $\underline{d}$ if $\underline{a}_{23} = 1$ [$\underline{a}_{23}=0, \underline{d}=(0,0,0,0,1,0,)$].

4. Bring the two halves of $\underline{b}$ together again in $\underline{o}$ after dropping the flag digits and adjusting the parity of each half. [$\underline{o} = (0,0,0,1,0,1,0,1,1,0)$]

5,6. Clear $\underline{c}$ register to zeros and decode the operation digits by inserting a 1 in the appropriate element of $\underline{c}$. [$\underline{c}_{26} \leftarrow 1$]

7. Increase the number represented by $\underline{a}$ by 2 and transfer the binary-coded-decimal equivalent to $\underline{A}^0$. [$\underline{A}^0 \Rightarrow 00402$].

8,9. Clear the $\underline{s}$ register and set the True/Complement Switch $\underline{s}_1 = 1$. (Used in addition).

10. Interchange the contents of $\underline{t}_1$ and $\underline{t}_2$ so that $\underline{t}_2$ is now in control.

State 2 ($\underline{t}_2=1$)

```
1   $\underline{a} \leftarrow \underline{A}^0$
2   $\underline{b} \leftarrow \underline{M}^{\underline{\beta}} \overset{+}{\underset{\times}{}} (\bar{\underline{\omega}}^1/\underline{a})$
3   $\underline{d} \leftarrow (\bar{\underline{a}}_{23}\,\underline{\varepsilon} \wedge \underline{\alpha}^6/\underline{b}) \vee (\underline{a}_{23}\underline{\varepsilon} \wedge \underline{\omega}^6/\underline{b})$
4   $\underline{r} \leftarrow (\underline{a}_{23}\underline{\varepsilon} \wedge ((\underline{b}_0 \neq \underline{b}_1),(2\downarrow\underline{\alpha}^4/\underline{b}))) \vee (\bar{\underline{a}}_{23}\underline{\varepsilon} \wedge ((\underline{b}_6 \neq \underline{b}_7),(\underline{\omega}^4/\underline{b})))$
                 $,(\underline{d}_0 \neq \underline{d}_1),(\underline{\omega}^4/\underline{d})$
5   $\underline{\alpha}^4/\underline{A}^4 \leftarrow \underline{\alpha}^4/\underline{A}^3 \leftarrow (\underline{r}_5 \neq \underline{r}_6),\underline{\omega}^3/\underline{r}$
6   $4\downarrow\underline{\alpha}^5/\underline{A}^4 \leftarrow 4\downarrow\underline{\alpha}^5/\underline{A}^3 \leftarrow \underline{\alpha}^5/\underline{r}$
7   $\underline{A}^0 \leftarrow \underline{\text{bcd}}\ (20000|(\underline{\gamma} \overset{+}{\underset{\times}{}} \underline{a} + 2))$
8   $\underline{t}_3 \leftrightarrow \underline{t}_2$
```

Fig. 5.6.4

1,2,3. As before. [$\underline{A}^0 \Rightarrow 00402, \underline{a}_{23}=0, \underline{d}=\underline{\alpha}^6/\underline{b}=(1,0,0,0,0,0) \Rightarrow 0 \quad \underline{\omega}^6/\underline{b} = (0,0,0,0,0,1) \Rightarrow 1$].

4. If $\underline{a}_{23}=0$, transfer the second half of $\underline{b}$ to the first half of $\underline{r}$; if $\underline{a}_{23} = 1$, transfer the first half of $\underline{b}$ to the first half of $\underline{r}$. Transfer $\underline{d}$ to the second half of $\underline{r}$. Drop off the flag digits and adjust parity in each half of $\underline{r}$. [$\underline{a}_{23}=0, \underline{r}=(0,0,0,0,1,1,0,0,0,0)$].

5. Transfer the second half of $\underline{r}$ to the first 4 digits of $\underline{A}^3$ and $\underline{A}^4$, dropping off $\underline{r}_6$ and adjusting parity. [$\underline{A}^3=\underline{A}^4 =(1,0,0,0,\ldots.) \Rightarrow$ 0XXXX].

6. Transfer the first half of $\underline{r}$ to $\underline{A}^3$ and $\underline{A}^4$. [$\underline{A}^3 = \underline{A}^4 =(1,0,0,0,0,0,0,0,1,\ldots) \Rightarrow$ 01XXX].

7. As before.

8. Transfer control to $\underline{t}_3$.

<u>State 3.</u> $\underline{t}_3 = 1$

| | |
|---|---|
| 1 | As in $\underline{t}_2$ |
| 2 | |
| 3 | |
| 4 | |
| 5 | $9\downarrow\underline{\alpha}^{10}/\underline{A}^3 \leftarrow 9\downarrow\underline{\alpha}^{10}/\underline{A}^4 \leftarrow \underline{\omega}^5/\underline{r}, \underline{\alpha}^5/\underline{r}$ |
| 6 | $\underline{A}^0 \leftarrow \underline{\text{bcd}}\ (20000 \,\vert\, (\underline{\gamma} \overset{+}{\times} \underline{a} + 2))$ |
| 7 | $\underline{t}_4 \leftrightarrow \underline{t}_3$ |

Fig. 5.6.5.

1,2,3,4. [$\underline{A}^0 \Rightarrow$ 00404, $\underline{a}_{23}=0, \underline{d}=\underline{\alpha}^6/\underline{b} = (1,0,0,1,0,1) \Rightarrow 5$, $\underline{\omega}^6/\underline{b} = (0,0,0,0,1,0) \Rightarrow 2, \underline{r} = (0,0,0,1,0,1,0,1,0,1)$ $\Rightarrow 25$]

5. Transfer the first half of $\underline{r}$ to $\underline{A}^3$ and $\underline{A}^4$ in positions $\underline{A}^i_{14}$ through $\underline{A}^i_{18}$; transfer second half of $\underline{r}$ to $\underline{A}^3$ and $\underline{A}^4$ in positions $\underline{A}^i_9$ through $\underline{A}^i_{13}$. [$\underline{A}^3=\underline{A}^4 \Rightarrow$ 0152X].

6. [$\underline{A}^0 \Rightarrow$ 00406].

7. Transfer control to $\underline{t}_4$.

<u>State 4.</u> $\underline{t}_4 = 1$

```
1  }
2  }    As in $\underline{t}_2$
3  }
4  }
5    $\underline{\omega}^5/\underline{A}^3 \leftarrow \underline{\omega}^5/\underline{A}^4 \leftarrow \underline{\omega}^5/\underline{r}$
6    $\underline{A}^0 \leftarrow \underline{\text{bcd}}\ (20000 | (\underline{\gamma} \overset{+}{\times} \underline{a} + 1))$
7    $\underline{t}_5 \leftrightarrow \underline{t}_4$
```

Fig. 5.6.6

1,2,3,4. [$\underline{A}^0 \Rightarrow$ 00406, $\underline{a}_{23}=0$, $\underline{\omega}^6/\underline{b} = (1,0,0,0,0,0) \Rightarrow 0$ (not used) $\underline{d} = \underline{\alpha}^6/\underline{b} = (0,0,0,1,1,1) \Rightarrow 7, \underline{r} = (1,0,0,0,0,0,0,1,1,1) \Rightarrow 07$] .

5. Transfer second half of $\underline{r}$ to last five positions of $\underline{A}^3$ and $\underline{A}^4$ [$\underline{A}^3 = \underline{A}^4 \Rightarrow 01527$]

6. [$\underline{A}^0 \Rightarrow 00407$]

<u>State 5.</u> $\underline{t}_5 = 1$

```
1
2      } As in $\underline{t}_2$
3
4
5      $(\underline{o}_1 \vee \underline{o}_2 \vee \underline{o}_3 \vee \bar{\underline{o}}_4)\underline{\epsilon} \wedge (\underline{\alpha}^4/\underline{A}^2) \leftarrow (\underline{r}_5 \neq \underline{r}_6), \underline{\omega}^3/\underline{r}$
6      $\underline{A}^0 \leftarrow \underline{bcd}\ (20000 \mid (\underline{\gamma} \overset{+}{\times} \underline{a}+2))$
7      $\underline{t}_6 \leftrightarrow \underline{t}_5$
```

Fig. 5.6.7

1,2,3,4. [$\underline{A}^0 \Rightarrow 00407$], $\underline{a}_{23} = 1$. $\underline{d} = \underline{\omega}^6/\underline{b} = (1,0,0,0,0,0) \Rightarrow 0$, $\underline{\alpha}^6/\underline{b} = (0,0,0,1,1,1) \Rightarrow 7$. $\underline{r} = (0,0,1,1,1,1,0,0,0,0) \Rightarrow 70$].

5. This transfer takes place only if $(\underline{o}_1 \vee \underline{o}_2 \vee \underline{o}_3 \vee \bar{\underline{o}}_4) = 1$, i.e., if the operation is not an immediate one. [Operation is <u>not</u> immediate. $\underline{A}^2 \Rightarrow$ 0XXXX].

6. [$\underline{A}^0 \Rightarrow 00409$].

$\underline{\text{State 6}}$ $\underline{t}_6=1$

| | |
|---|---|
| 1 | $\left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\}$ As in $\underline{t}_2$ |
| 2 | |
| 3 | |
| 4 | |
| 5 | $(\underline{o}_1 \vee \underline{o}_2 \vee \underline{o}_3 \vee \underline{\bar{o}}_4)\underline{\varepsilon} \wedge (4 \downarrow \underline{\alpha}^{10}/\underline{A}^2) \leftarrow \underline{r}$ |
| 6 | $\underline{s}_9 \leftarrow \underline{i}(\underline{\delta} \overset{+}{\times} \underline{r}) \wedge (\underline{c}_{46} \vee \underline{c}_{47})$ |
| 7 | $\underline{A}^0 \leftarrow \underline{bcd}\ (20000 \mid (\underline{\gamma} \overset{+}{\times} \underline{a} + 2))$ |
| 8 | $\underline{t}_7 \leftrightarrow \underline{t}_6$ |

Fig. 5.6.8

1,2,3,4. [$\underline{A}^0 \Rightarrow 00409$, $\underline{a}_{23} = 1$, $\underline{d} = \underline{\omega}^6/\underline{b} = (1,0,0,1,0,1) \Rightarrow 5$, $\underline{\alpha}^6/\underline{b} = (0,0,0,0,0,1) \Rightarrow 1$, $\underline{r} = (0,0,0,0,1,1,0,1,0,1) \Rightarrow 15$.]

5. [Not immediate. $\underline{A}^2 \Rightarrow 015XX$].

6. If the operation is 46 or 47 and if the indicator denoted by the contents of $\underline{r}$ has the value 1, set the Branch Test switch, $\underline{s}_9$. [Not applicable.]

7. [$\underline{A}^0 \Rightarrow 00411$].

State 7, $\underline{t}_7 = 1$

1 $\left.\begin{array}{l} \\ \\ \\ \end{array}\right\}$ As in $\underline{t}_2$
2
3
4
5 $(\underline{o}_1 \vee \underline{o}_2 \vee \underline{o}_3 \vee \bar{\underline{o}}_4)\underline{\varepsilon} \wedge (\underline{\omega}^{10}/\underline{A}^2) \leftarrow \underline{r}$
6 $\underline{A}^0 \leftarrow \underline{bcd}\ \ (20000 \mid (\underline{\gamma} \overset{+}{\times} \underline{a} + 1))$
7 $\underline{t}_8 \leftrightarrow \underline{t}_7$

Fig. 5.6.9

1,2,3,4. [$\underline{A}^0 \Rightarrow 00411, \underline{a}_{23}=1, \underline{d}=\underline{\omega}^6/\underline{b} = (1,0,0,0,1,1) \Rightarrow 3,$
$\underline{\alpha}^6/\underline{b}=(0,0,0,0,1,0) \Rightarrow 2, \underline{r} = (0,0,0,1,0,1,0,0,1,1) \Rightarrow 23.$]

5. [Not immediate. $\underline{A}^2 \Rightarrow 01523$].

State 8. $\underline{t}_8 = 1$

1 $(\bar{\underline{o}}_1 \wedge \bar{\underline{o}}_2 \wedge \bar{\underline{o}}_3 \wedge \underline{o}_4)\underline{\varepsilon} \wedge \underline{A}^2 \leftarrow \underline{a}$
2 $\underline{s}_1 \leftarrow (\overline{\underline{c}_{12} \vee \underline{c}_{22} \vee \underline{c}_{14} \vee \underline{c}_{24}})$
3 $\underline{i}_{10} \leftarrow \underline{i}_{11} \leftarrow \underline{i}_{12} \leftarrow \underline{s}_{10} \leftarrow 1$
4 $\underline{t}_{26} \leftarrow \underline{c}_{25} \vee \underline{c}_{26} \vee \underline{c}_{31} \vee \underline{c}_{15} \vee \underline{c}_{16}$
$\underline{t}_{11} \leftarrow \underline{c}_{21} \vee \underline{c}_{22} \vee \underline{c}_{24} \vee \underline{c}_{11} \vee \underline{c}_{12} \vee \underline{c}_{14}$
$\underline{t}_{32} \leftarrow \underline{c}_{23} \vee \underline{c}_{13}$
$\underline{t}_{15} \leftarrow \underline{c}_{27} \vee \underline{c}_{17}$
$\underline{t}_{20} \leftarrow \underline{c}_{42}$
$\underline{t}_{28} \leftarrow \underline{c}_{32} \vee \underline{c}_{33} \vee \underline{c}_{43} \vee \underline{c}_{44} \vee \underline{c}_{45}$
$\underline{t}_{18} \leftarrow (\underline{c}_{46} \wedge \underline{s}_9) \vee (\underline{c}_{47} \wedge \bar{\underline{s}}_9) \vee \underline{c}_{49}$
$\underline{t}_1 \leftarrow (\underline{c}_{46} \wedge \bar{\underline{s}}_9) \vee (\underline{c}_{47} \wedge \underline{s}_9) \vee \underline{c}_{41}$
$\underline{t}_8 \leftarrow 0$

Fig. 5.6.10

1. If operation is immediate insert address of last digit of the instruction into $\underline{A}^2$. If not immediate no transfer takes place. [Not immediate.]
2. Turn True/Complement switch on Complement (0) if operation is subtract or compare. [$\underline{s}_1 \leftarrow 1$].
3. Set indicators and First Cycle switch.
4. Transfer control to appropriate state depending on operation to be performed. [$\underline{t}_{26} \leftarrow 1$].

<u>State 26</u>. $\underline{t}_{26} = 1$

| | |
|---|---|
| 1 | $\underline{a} \leftarrow \underline{A}^2$ |
| 2 | $\underline{b} \leftarrow \underline{M}^{\underline{\beta}} \overset{+}{\times} (\bar{\underline{\omega}}^1 / \underline{a})$ |
| 3 | $\underline{d} \leftarrow (\bar{\underline{a}}_{23}\, \underline{\varepsilon} \wedge \underline{\alpha}^6 / \underline{b}) \vee (\underline{a}_{23} \underline{\varepsilon} \wedge \underline{\omega}^6 / \underline{b})$ |
| 4 | $\underline{A}^2 \leftarrow \underline{bcd}\ (20000 \vert (\underline{\gamma} \overset{+}{\times} \underline{a} + \underline{c}_{31} - (\underline{c}_{15} \vee \underline{c}_{25} \vee \underline{c}_{16} \vee \underline{c}_{26} \vee \underline{c}_{17} \vee \underline{c}_{27}))$ |
| 5 | $\underline{s}_4 \leftarrow \bar{\underline{s}}_{10} \wedge \underline{d}_1$ |
| 6 | $\underline{s}_6 \leftarrow \underline{d}_2 \wedge \bar{\underline{d}}_3 \wedge \underline{d}_4 \wedge \bar{\underline{d}}_5$ |
| 7 | $\underline{t}_{27} \leftrightarrow \underline{t}_{26}$ |

Fig. 5.6.11

1,2. Transfer from $\underline{M}$ the 12 digit vector defined by the address in $\underline{A}^2$. [$\underline{A}^2 \Rightarrow 01523$, $\underline{b} = (0,0,1,0,0,0,0,1,1,0,0,1) \Rightarrow 8\bar{9}$].

3. Copy the first half of $\underline{b}$ into $\underline{d}$ if $\underline{a}_{23}=0$. Copy the second half of $\underline{b}$ into $\underline{d}$ if $\underline{a}_{23}=1$. [$\underline{a}_{23}=1, \underline{d}=(0,1,1,0,0,1) \Rightarrow \bar{9}$].
4. If the operation if Transmit Record, add 1 to the contents of $\underline{a}$ and place the result in $\underline{A}^2$. If the operation is Transmit Digit or Transmit Field, or Branch and Transmit, subtract 1 from the contents of $\underline{a}$ and place the result in $\underline{A}^2$. [Transmit Field. $\underline{A}^2 \Rightarrow 01522$].
5. If it is not the first cycle and there is a flag digit in $\underline{d}$ ($\underline{d}_1$) set $\underline{s}_4=1$($\underline{s}_4$ is Field Mark No. 1 Switch.) [$\underline{s}_4 = 0$].
6. If the vector in $\underline{d}$ represents a Record Mark ($\ddagger \Rightarrow (1,0,1,0,1,0)$ set $\underline{s}_6=1$. ($\underline{s}_6$ is Record Mark Switch.) [$\underline{s}_6=0$].
7. Transfer Control to state 27.

<u>State 27</u>. $\underline{t}_{27} = 1$

```
1   $\underline{a} \leftarrow \underline{A}^3$
2   $\underline{b} \leftarrow \underline{M}^{\underline{\beta}} \overset{+}{\times} (\bar{\underline{\omega}}^1/\underline{a})$
3   $\underline{A}^3 \leftarrow \underline{bcd}\ (20000 | \gamma^{+}_{x}\underline{a}+\underline{c}_{31}-(\underline{c}_{15} \vee \underline{c}_{25} \vee \underline{c}_{16} \vee \underline{c}_{26} \vee \underline{c}_{17} \vee \underline{c}_{27}))$
4   $\underline{s}_{10} \leftarrow 0$
5   $\underline{(\bar{\underline{a}}_{23}\ \underline{\varepsilon} \wedge \underline{\alpha}^6/\underline{b}) \vee (\underline{a}_{23}\underline{\varepsilon} \wedge \underline{\omega}^6/\underline{b})}$
    $\leftarrow \neq/((\bar{\underline{s}}_{12} \wedge \underline{s}_{13}) \vee (\bar{\underline{s}}_{12} \wedge \bar{\underline{s}}_{13} \wedge \underline{d}_1), \underline{\omega}^4/\underline{d}), (\bar{\underline{s}}_{12} \wedge \underline{s}_{13}) \vee (\bar{\underline{s}}_{12} \wedge \bar{\underline{s}}_{13} \wedge \underline{d}_1),$
    $\underline{\omega}^4/\underline{d}$
6   $\underline{M}^{\underline{\beta}} \overset{+}{\times} (\bar{\underline{\omega}}^1/\underline{a}) \leftarrow \underline{b}$
7   $\underline{t}_{26} \leftarrow (\underline{c}_{31} \wedge \bar{\underline{s}}_6) \vee ((\underline{c}_{16} \vee \underline{c}_{26} \vee \underline{c}_{17} \vee \underline{c}_{27}) \wedge \bar{\underline{s}}_4)$
    $\underline{t}_1 \leftarrow (\underline{c}_{31} \wedge \underline{s}_6) \vee ((\underline{c}_{16} \vee \underline{c}_{26} \vee \underline{c}_{17} \vee \underline{c}_{27}) \wedge \underline{s}_4) \vee\ \underline{c}_{25} \vee \underline{c}_{15}$
```

Fig. 5.6.12

1,2. Transfer from $\underline{M}$ the 12-digit vector defined by the address in $\underline{A}^3$. [$\underline{A}^3 \Rightarrow 01527$, $\underline{b} = (0,1,0,0,0,0,1,0,0,1,1,0) \Rightarrow \bar{0}6$].

3. As in state 26, (4). [Transmit Field. $\underline{A}^3 \Rightarrow 01526$].

4. Set First Cycle switch to 0.

5. Transfer the last 4 digits of $\underline{d}$ to either the first or last half of $\underline{b}$ depending on whether $\underline{a}_{23}=0$ or 1 respectively. If a flag is to be cleared ($\underline{s}_{12}=1$) or if $\underline{d}_1=0$ and $\underline{s}_{12}=\underline{s}_{13}=0$, set $\underline{b}_1$ or $\underline{b}_7=0$ ($\underline{b}_1$, if $\underline{a}_{23}=0$; $\underline{b}_7$, if $\underline{a}_{23}=1$). If a flag is to be set ($\underline{s}_{13}=1$) or if $\underline{d}_1=1$ and $\underline{s}_{12}=\underline{s}_{13}=0$, set $\underline{b}_1$ or $\underline{b}_7=1$, ($\underline{b}_1$, if $\underline{a}_{23}=0$; $\underline{b}_7$, if $\underline{a}_{23}=1$). Adjust parity of transferred digits. [$\underline{s}_{12}=\underline{s}_{13}=0$; $\underline{a}_{23}=1$; $\underline{\omega}^6/\underline{b} \leftarrow \underline{d}=(0,1,1,0,0,1) \Rightarrow \bar{9}$; $\underline{b} = (0,1,0,0,0,0,0,1,1,0,0,1) \Rightarrow \bar{0}\bar{9}$.]

6. Transfer $\underline{b}$ to Memory at address given by $\underline{a}$.

7. Go to state 26 if the operation is Transmit Record (31) and the Record Mark switch is off ($\underline{s}_6=0$) or if the operation is Transmit Field or Branch and Transmit and the Field Mark No. 1. switch is off. ($\underline{s}_4=0$)
   Go to state 1 if the operation is Transmit Record and the Record Mark Switch is ON ($\underline{s}_6=1$) or if the operation is Transmit Field or Branch and Transmit and the Field Mark No. 1 switch is ON, ($\underline{s}_4=1$), or if the operation is Transmit Digit.

[$\underline{s}_4$=0, $\underline{c}_{26}$=1, go to state 26].

State 26. (Second Cycle)

1,2. [$\underline{A}^2 \Rightarrow$ 01522, $\underline{b}$=(1,1,0,0,0,1,0,0,1,0,0,0) $\Rightarrow \bar{1}8$]

3. [$\underline{a}_{23}$= 0, $\underline{d}$=(0,0,1,0,0,0,) $\Rightarrow$ 8].

4. [$\underline{A}^2 \Rightarrow$ 01521].

5,6. [$\underline{d}_1$=0; $\underline{s}_{10}$=0; $\underline{s}_4 \leftarrow$ 0].

State 27. (Second Cycle)

1,2. [$\underline{A}^3 \Rightarrow$ 01526, $\underline{b}$=(0,1,0,0,0,0,0,1,1,0,0,1) $\Rightarrow \bar{0}\bar{9}$].

3. [$\underline{A}^3 \Rightarrow$ 01525].

5. [$\underline{a}_{23}$=0, $\underline{s}_{12}$=$\underline{s}_{13}$=0; $\underline{\alpha}^6/\underline{b} \leftarrow \underline{d}$ = (0,0,1,0,0,0) $\Rightarrow$ 8;
   $\underline{b}$=(0,0,1,0,0,0,0,0,1,1,0,0,1) $\Rightarrow 8\bar{9}$].

7.[$\underline{s}_4$=0, $\underline{c}_{26}$=1; go to state 26.]

State 26. (Third Cycle)

1,2. [$\underline{A}^2 \Rightarrow$ 01521, $\underline{b}$ = (0,0,0,1,0,0,1,1,0,0,0,1) $\Rightarrow 4\bar{1}$].

3. [$\underline{a}_{23}$=1, $\underline{d}$ = (1,1,0,0,0,1) $\Rightarrow \bar{1}$]

4. [$\underline{A}^2 \Rightarrow$ 01520]

5,6. [$\underline{d}_1$ = 1; $\underline{s}_{10}$ = 0; $\underline{s}_4 \leftarrow$ 1].

State 27. (Third Cycle)

1,2. [$\underline{A}^3 \Rightarrow 01525$, $\underline{b} = (0,0,0,0,0,1,1,1,0,1,0,0) \Rightarrow 1\bar{4}$].

3. [$\underline{A}^3 \Rightarrow 01524$].

5. [$\underline{a}_{23} = 1$, $\underline{s}_{12} = \underline{s}_{13} = 0$; $\underline{\omega}^6/\underline{b} \leftarrow \underline{d} = (1,1,0,0,0,1) \Rightarrow \bar{1}$;

$\underline{b} = (0,0,0,0,0,1,1,1,0,0,0,1) \Rightarrow 1\bar{1}$].

7. [$\underline{s}_4 = 1$, $\underline{c}_{26} = 1$; go to state 1.]

## Section 5.7 Concluding Remarks

By now, the often-repeated remark that a digital computer is a complicated device will have acquired some significance. Yet, by human standards, it seems to accomplish very little. In particular, we have just seen that the 1620 had to perform something like 110 transfers of information to interpret and execute an instruction. Whether we accept this complexity as inevitable or decide that computers can be made to perform the same tasks in a simpler way, we are still faced with analysing a very intricate organisation to obtain any understanding of computers in general.

The two main tools we have used are directed graphs and an algorithmic language. The directed graphs of the static structure (Fig. 5.3.1) and the dynamic structure (Fig. 5.5.5) provide a simple way of visualising the gross operation of the machine and the algorithmic language provides a detailed mathematical description. Both of these tools are capable of further development and this will be discussed in Chapter 6.

A final word on the 1620 seems appropriate. As has been mentioned before, the 1620 structure is not typical of current computer organisations. The complexity of its structure, of itself, suggests that some means of visualising it be formed (such as Fig. 5.5.5). While the newer computers may not have this particular structure, they almost certainly will be at least as complex and tools such as directed graphs will be very useful in understanding them.

The study of the 1620 also points up the large gap which exists between the language the programmer uses and the way the machine implements the language. Normally, the programmer learns a series of seemingly unrelated tricks which help to make his programs more efficient. These tricks are not at all obvious in the programming language but are usually made clear by a reasonable under-

standing of the structure of the machine. Hence, it is evident that a language with sufficient power to describe a machine at a detailed level and still be useful as a programming language, would contribute a great deal to the efficient use of machines.

## Appendix 5.a Static Transfers of the IBM 1620

Special vectors:

$\underline{\varphi} = (0,8,4,2,1)$

$\underline{\delta} = (10\underline{\varphi}, \underline{\varphi})$

$\underline{\gamma} = (10000(\downarrow \underline{\bar{\alpha}}^{1}/\underline{\varphi}),\ 1000\underline{\varphi},\ 100\underline{\varphi},\ 10\underline{\varphi},\ \underline{\varphi})$

$\underline{\beta} = \underline{\bar{\omega}}^{1}/(\cdot 5\underline{\gamma})$

Special Function

<u>bcd</u>(x) is the <u>vector</u> representing the 1620 binary-coded-decimal equivalent of the integer x, each decimal digit being encoded in the form (C,8,4,2,1). (See Appendix 5.b.). The dimension of the vector is determined by the destination of the transfer in which it appears.

<u>Transfer to Memory</u> ($\underline{M}$)

$$\boxed{\underline{M}^{\underline{\beta}} \overset{+}{\times} (\bar{\underline{\omega}}^{1}/\underline{a}) \leftarrow \underline{b}}$$

<u>Transfer to Memory Data Register</u> ($\underline{d}$)

$$\boxed{\underline{d} \leftarrow (\bar{\underline{a}}_{23}\underline{\varepsilon} \wedge \underline{\alpha}^{6}/\underline{b}) \vee (\underline{a}_{23}\underline{\varepsilon} \wedge \underline{\omega}^{6}/\underline{b})}$$

<u>Transfers to Memory Buffer Register</u> ($\underline{b}$)

$$\boxed{\begin{array}{l}
\qquad\qquad\qquad\qquad \underline{b} \leftarrow \underline{M}^{\underline{\beta}} \overset{+}{\times} (\bar{\underline{\omega}}^{1}/\underline{a}) \\[1em]
(\bar{\underline{a}}_{23}\underline{\varepsilon} \wedge \underline{\alpha}^{6}/\underline{b}) \vee (\underline{a}_{23}\underline{\varepsilon} \wedge \underline{\omega}^{6}/\underline{b}) \leftarrow \\
\hline
\quad \neq/((\bar{\underline{s}}_{12} \wedge \underline{s}_{13}) \vee (\bar{\underline{s}}_{12} \wedge \bar{\underline{s}}_{13} \wedge \underline{d}_{1}), \underline{\omega}^{4}/\underline{d}), (\bar{\underline{s}}_{12} \wedge \underline{s}_{13}) \\
\qquad\qquad\qquad \vee (\bar{\underline{s}}_{12} \wedge \bar{\underline{s}}_{13} \wedge \underline{d}_{1}), \underline{\omega}^{4}/\underline{d} \\[1em]
\qquad\qquad\qquad \underline{b} \leftarrow \underline{bcd}\ (0)
\end{array}}$$

Transfers to Memory Address Register $\underline{a}$

$$\underline{a} \leftarrow \underline{A}^i$$

$$\underline{\omega}^5/\underline{a} \leftarrow (\underline{s_1\varepsilon} \wedge \underline{\omega}^5/\underline{r}) \vee (\underline{\bar{s}_1\varepsilon} \wedge \underline{bcd}(10 | (9 - \underline{\varphi} \overset{+}{\times} \underline{\omega}^5/\underline{r} + \underline{s}_7)))$$

$$\underline{a} \leftarrow \underline{bcd}\ (80)$$

$$\underline{a} \leftarrow \underline{bcd}\ (300)$$

$$\underline{a} \leftarrow \underline{bcd}\ (0)$$

$$((10 \uparrow \underline{\omega}^5) \vee \underline{\omega}^5)/\underline{a} \leftarrow \underline{bcd}\ (10 + 2 \times (\underline{\varphi} \overset{+}{\times} \underline{m}))$$

$$5 \uparrow \underline{\omega}^5/\underline{a} \leftarrow (\underline{\bar{s}_3\varepsilon} \wedge ((\underline{d}_0 \neq \underline{d}_1), \underline{\omega}^4/\underline{d})) \vee (\underline{s_3\varepsilon} \wedge \underline{\alpha}^1)$$

Transfers to Memory Address Register Storage $\underline{A}$

$$\underline{A}^i \leftarrow \underline{bcd}\ (20000 | (\underline{\gamma} \overset{+}{\times} \underline{a} + j)),\quad j = -1, +1, +2.$$

$$\underline{A}^i \leftarrow \underline{a}$$

$$\underline{\alpha}^4/\underline{A}^i \leftarrow (\underline{r}_5 \neq \underline{r}_6),\ \underline{\omega}^3/\underline{r}$$

$$4 \downarrow \underline{\alpha}^5/\underline{A}^i \leftarrow \underline{\alpha}^5/\underline{r}$$

$$9 \downarrow \underline{\alpha}^5/\underline{A}^i \leftarrow \underline{\omega}^5/\underline{r}$$

$$14 \downarrow \underline{\alpha}^5/\underline{A}^i \leftarrow \underline{\alpha}^5/\underline{r}$$

$$\underline{\omega}^5/\underline{A}^i \leftarrow \underline{\omega}^5/\underline{r}$$

Transfers to Digit Register $\underline{r}$

$$\boxed{\begin{aligned}
&\underline{r} \leftarrow (\underline{a}_{23}\underline{\epsilon} \wedge ((\underline{b}_0 \neq \underline{b}_1),(2 \downarrow \underline{\alpha}^4/\underline{b})))\\
&\qquad \vee(\underline{\bar{a}}_{23}\underline{\epsilon} \wedge ((\underline{b}_6 \neq \underline{b}_7),\underline{\omega}^4/\underline{b})),(\underline{d}_0 \neq \underline{d}_1),\underline{\omega}^4/\underline{d}\\
&\quad \underline{\omega}^5/\underline{r} \leftarrow \underline{\alpha}^5/\underline{r}\\
&\underline{\omega}^5/\underline{r} \leftarrow (\underline{\bar{s}}_3\underline{\epsilon} \wedge \underline{\alpha}^1) \vee (\underline{s}_3\underline{\epsilon} \wedge \underline{\omega}^5/\underline{r})
\end{aligned}}$$

Transfer to Operation Register $\underline{o}$

$$\boxed{\underline{o} \leftarrow (\underline{d}_0 \neq \underline{d}_1),(\underline{\omega}^4/\underline{d}),(\underline{b}_6 \neq \underline{b}_7),(\underline{\omega}^4/\underline{b})}$$

Transfer to Multiplier Register $\underline{m}$

$$\boxed{\underline{m} \leftarrow (\underline{d}_0 \neq \underline{d}_1),(\underline{\omega}^4/\underline{d})}$$

[illegible]

Appendix 5.b 1620 Binary-Coded Decimal

| | C | F | 8 | 4 | 2 | 1 | | C | F | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | $\bar{0}$ | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | $\bar{1}$ | 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 | $\bar{2}$ | 1 | 1 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 0 | 1 | 1 | $\bar{3}$ | 0 | 1 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 | $\bar{4}$ | 1 | 1 | 0 | 1 | 0 | 0 |
| 5 | 1 | 0 | 0 | 1 | 0 | 1 | $\bar{5}$ | 0 | 1 | 0 | 1 | 0 | 1 |
| 6 | 1 | 0 | 0 | 1 | 1 | 0 | $\bar{6}$ | 0 | 1 | 0 | 1 | 1 | 0 |
| 7 | 0 | 0 | 0 | 1 | 1 | 1 | $\bar{7}$ | 1 | 1 | 0 | 1 | 1 | 1 |
| 8 | 0 | 0 | 1 | 0 | 0 | 0 | $\bar{8}$ | 1 | 1 | 1 | 0 | 0 | 0 |
| 9 | 1 | 0 | 1 | 0 | 0 | 1 | $\bar{9}$ | 0 | 1 | 1 | 0 | 0 | 1 |
| $\ddagger$ | 1 | 0 | 1 | 0 | 1 | 0 | $\bar{\ddagger}$ | 0 | 1 | 1 | 0 | 1 | 0 |

C = check position (odd parity).

F = flag position.

8,4,2,1 are weights associated with other positions.

This is the code used in Memory, Memory Buffer Register and Memory Data Register. In other parts of the machine, 4 or 5 binary digits are used. In these the representation is C,8,4,2,1 or C,4,2,1.

# CHAPTER 6

# Conclusions.

## Section 6.1 Information and Computers

Although the three main fields surveyed in this thesis, information theory, coding theory and digital computers are related, they have developed along rather independent lines. The interaction between the study of computers and the other two fields is quite small and tends to be limited to such generalisations as "computers are information-processing machines." Rather than attempt to summarise three rather isolated topics, we shall devote this section to pointing out some of the reasons for lack of interaction between the fields.

The common core of information theory, coding theory and the study of computers is, of course, the concept of information. Information theory makes the important contribution of concentrating attention on the statistical properties of information. From these properties, it has developed constructive - but somewhat impractical - encoding schemes. It has stimulated studies on the synthesis of reliable machines from unreliable (noisy) components

(DeLeener et al., von Neumann). The concepts of information theory give new insight into the design and operation of computers and it may well be that computers can be organised to permit the application of these concepts, e.g., by simultaneous operation on larger blocks of information or by a more penetrating analysis of the algorithms of computation. Yet there are many difficulties in the way of this extension of information theory concepts which are not present in the original problem of simple communication. For example, the memory of a computer, which may hold about $10^6$ binary digits, seems to offer the possibility of applying encoding principles whose effectiveness depends on handling long sequences of interrelated digits. As we have seen in the 1620, computers interpret and execute instructions by selecting short sequences of digits from memory and distributing them to various registers of the machine to each of which error control would have to applied independently. The fragmentation of long sequences forces the designer to rely on the least efficient methods of error control - the encoding of short sequences.

The difficulties encountered in finding a suitable encoding scheme are typical of the study of computers in general: it is seldom possible to solve a particular

problem without taking into account restrictions imposed by many other considerations. Very little is known about such complex structures as computers and it would be mere speculation to suggest how information theory could be applied to them. However, the application of algorithmic languages and directed graphs to the study of computers is less uncertain and hopefully may lead among other things, to a greater understanding of the relationship between information theory and machines. The development of these two topics is the subject of the next two sections.

## Section 6.2 Algorithmic Languages

The current interest in algorithmic languages stems from the fact that algorithms written in a well-defined langugage can be translated automatically into machine language by a machine. As a result, most of the algorithmic languages in use have limited themselves to the small set of characters or symbols available on current input-output devices and to expressing a statement on single lines of type. These technical considerations have restricted the languages quite severely and rather unnecessarily. They are forced to abandon standard mathematical notation (which depends on a two-dimensional presentation)

and, consequently to abandon the ability of a good notation to bring out essential ideas and suggest generalisations.

The languages are also influenced by the nature of current machines which normally operate on one field at a time. While the languages make use of a vector and matrix reference system, they require that operations on vectors and matrices be displayed as explicit operations on the individual elements. The overall effect is that the essential points of algorithms described in these languages disappear in a mass of "housekeeping" details.

While these characteristics tend to make the translation process easier, they require that the user do his thinking aided by a conventional mathematical notation and then translate into an algorithmic language. This alone would make it desirable to combine mathematical notation and algorithmic languages. The language of Iverson [1] clearly demonstrates the remarkable generalisations that are possible when such a combination is attempted. The language in Chapter 5 is a small subset of the main language which is capable of describing in a uniform way sequential processes of widely differing characters. Aside from its importance as an algorithmic language, it suggests a number of ways in which machines might be changed to make them more efficient in executing algorithms.

Algorithmic languages already have had an effect on the design of digital computers. A number of machines have appeared which are organised to facilitate the translation from an algorithmic language to a machine language (Burroughs B5000, Ferranti Atlas, English Electric KDF-9, etc.) The translation process involves scanning an algorithmic statement such as $c \leftarrow a(b + 1.3)$ to pick out operators and operands and constructing a sequence of instructions to carry out the function. The statement $c \leftarrow a(b + 1.3)$ may be thought of as a vector whose elements happen to be symbols (literals) and the facilities provided by more advanced machines for manipulating a string of literals <u>as a whole</u>, are actually physical realisations of elementary operations on vectors. These manipulations tend to be somewhat specialised but they demonstrate that more general operations on vectors are feasible. The algorithmic language of Chapter 5 suggests what form these general operations should take, e.g., the more important manipulations would be

$$\underline{x} \odot \underline{y}$$

$$\odot/\underline{x} \quad \text{(Compression)}$$

$$\underline{u}/\underline{x} \quad \text{(Selection)}$$

where $\odot = \leftarrow, +, \times,$ etc.

(Some of these vector operations appear in current machines in rudimentary form; for example,

$$\underline{x} \leftarrow \underline{y}$$

can be carried out in the 1620 by means of a Transmit Record (31) instruction, if the elements of $\underline{y}$ are represented as a consecutive set of fields followed by a record mark.)

The language suggests two other possible developments. The first is reading a typed or handwritten document directly - this would permit a machine to accept programs written in a two-dimensional notation and eliminate the awkward transformation of notation necessary with present input equipment. Such devices are currently in operation but are not widely used because of their cost. The other development (which would be even more costly) is that of using a set of elementary computers to operate simultaneously on the elements of a vector or matrix. A matrix of computers has been suggested as the solution to the problem of solving partial differential equations but would be useful in connection with algorithmic languages.

The use of even elementary algorithmic languages has proved to be so successful in practice that it is hardly necessary to predict that they will have a major influence on the development and use of computers.

## Section 6.3 Directed Graphs and Seuqential Machines

Directed graphs occur in many forms in connection with sequential processes. Computer programmers are well acquainted with them in the form of flow-charts which display the gross sequences of steps in an algorithm. We have already seen how they help in visualising the static and dynamic structure of a machine (Fig. 5.3.1 and 5.5.5.) They are, of course, interesting in their own right from the view-point of graph theory but our interest here lies in their application to the study of finite sequential machines.

The study of finite sequential machines leads quickly into highly abstract mathematics and we will do no more than suggest the general lines of enquiry by giving a simple example. Consider a machine whose function is to produce the sum of two binary numbers which will be presented to the machine as a sequence of pairs of binary digits. As each pair of digits is received, the machine computes and outputs each sum digit. The machine must have at least one binary storage element since a carry from one pair of input digits must be held until the next pair are available. The machine must do two things: it must output a 0 or a 1 depending on the input pair of digits and the presence or absence of a carry and it must set its memory element to

indicate a carry or no carry for the next pair.

There are 8 possible combinations of input digits and carries and they may be analysed as follows:

Let $c_0$ be the state of the memory element which indicates no carry and $c_1$ the state indicating a carry. Then, if the element is in state $c_0$(no carry) and the pair (0,0), (0,1), or (1,0) is received, the element remains in the same state but if the pair (1,1) is received, the element must be changed to state $c_1$. Similarly, if the element is in state $c_1$ and the pair (1,0), (0,1) or (1,1) is received, then it stays in state $c_1$(carry) but if (0,0) is received then it is changed state to $c_0$. A directed graph or equivalently, a state table puts this discussion in a more useful form. Each of the eight possible

$$\begin{tikzcd}
c_0 \arrow[loop left] \arrow[rr, bend right] & & c_1 \arrow[loop right] \arrow[ll, bend right]
\end{tikzcd}$$

<u>State Diagram</u>

| | | (0,0) | (0,1) | (1,0) | (1,1) |
|---|---|---|---|---|---|
| Present State | $c_0$ | $c_0$ | $c_0$ | $c_0$ | $c_1$ |
| | $c_1$ | $c_0$ | $c_1$ | $c_1$ | $c_1$ |

<u>Next State Table</u>

|  |  | Input Pairs |  |  |  |
|---|---|---|---|---|---|
|  |  | (0,0) | (0,1) | (1,0) | (1,1) |
| Present | $c_0$ | 0 | 1 | 1 | 0 |
| State | $c_1$ | 1 | 0 | 0 | 1 |

<u>Output Value Table</u>

Fig. 6.2.1

combinations gives rise to an output value 0 or 1 as indicated in the Output Value Table.

The function of the machine is completely determined by the Output Value Table and all machines which realises this table are said to be <u>equivalent</u>. Clearly, there are many other machines which realise this table and the purpose of such an analysis is to discover which of these is minimal in some sense, e.g., in the number of memory elements. If the function to be realised is even a little more complicated than this, the problem of analysis is quite difficult. Certain minimisation techniques have been devised but the overall economisation of a machine like the 1620 is still beyond reach of the present theory.

The short analysis given above provides a useful

definition of the concept of "state" which was used rather loosely in Chapter 5. The states of a machine may be thought of as the combination of states of the storage elements which comprise the machine. The theory seeks to analyse the totality of states in which a machine can exist. Clearly, in a machine with as many storage elements as a computer, the number of states is exceedingly large and powerful methods will be needed to make any analysis meaningful.

However, the theory confers at least one subsidiary benefit; it suggests that the study of machines (especially computers) should be divorced from ideas of "function" and "purpose." While such ideas are helpful in understanding the operation of digital computers, in the long run, it is likely that they will act as a hindrance rather than a help to the development of an overall theory of machines. An analogy may be drawn between the present situation of the theory and that of information theory before 1948. Little progress was made there until information was rid of connotations of "meaning" and "value" and could be clearly defined as a probabilistic concept.

## Section 6.4 Concluding Remarks

It should be clear by now that the study of

computers requires more than a cursory acquaintance with many fields of mathematics, e.g., probability theory, modern algebra, graph theory, number theory, etc. Like other eclectic fields it suffers from a lack of communication between the workers in the field and, consequently, much haphazard and sometimes duplicated effort. Because so many important aspects of computers have scarcely been touched it is very difficult, if not impossible, to see how the subject will develop but it is clear that a great deal of effort must go into examining how the various disciplines involved are related to each other. The enormous theoretical and practical significance of computers provides a powerful motive for such an effort.

## BIBLIOGRAPHY

ABRAMSON, N.M. [1], A Class of Systematic Codes for Non-Independent Errors. Technical Report No. 51, Dec. 30, 1958, Stanford Electronic Laboratories, Stanford, Calif.

ABRAMSON, N.M., [2], Error-Correcting Codes from Linear Sequential Circuits, in "Information Theory, Fourth London Symposium," Colin Cherry, (Ed.), Butterworths, London, 1961.

ANDREE, R.V., "Selections from Modern Abstract Algebra," Holt, Rinehart and Winston, Inc., New York, 1958.

BARTEE, T.C., LEBOW, I.L., REED, I.S., "Theory and Design of Digital Machines," McGraw-Hill, 1962.

Bemer, R.W., The American Standard Code for Information Exchange, Datamation, Aug. 1963.

BRILLOUIN, L., "Science and Information Theory," Academic Press, New York, N.Y., 2nd. Edition (1962).

BROOKS, F.P., BLAAUW, G.A., and BUCHHOLZ, W., Processing Data in Bits and Pieces, IRE Trans. on Electronic Computers, Vol. EC-8, p 118-124, (1959).

BUCHHOLZ, W., (Ed.), "Planning a Computer System," McGraw-Hill, New York, 1962.

CHERRY, C., "On Human Communication," Science Editions Inc., New York, 1961.

DeLEENER, R.E., MOORE, E.F., SHANNON, C.E., and SHAPIRO, N., Computability by Probabilistic Machines, "Automata Studies," Ann. of Math. Studies, No. 34, Princeton Univ. Press, Princeton, N.Y., 1956.

DIMSDALE, B., and WIENBERG, G.M., Programmed Error Correction for Project Mercury, Comm. of A.C.M., Vol. 3, 12, (1960).

ELIAS, P., [1], Error-Free Coding, IRE Trans., PGIT-4, 29-37, (1954).

ELIAS, P., [2], Coding and Decoding, in "Lectures on Communication System Theory," Edited by E.J. Baghdady, McGraw-Hill, New York, 1961.

ELSPAS, B., The Theory of Autonomous Linear Sequential Networks, IRE Trans. on Circuit Theory, CT-6, No. 1, 45-60, (1959).

FANO, R.M., [1] Conclusion: Present Trends, in "Lectures on Communication System Theory," Edited by E.J. Baghdady, McGraw-Hill, New York, 1961.

FANO, R.M., [2], "Transmission of Information," M.I.T. Press and John Wiley, Inc., New York, 1961.

FEINSTEIN, A., "Foundations of Information Theory," McGraw-Hill, New York, 1958.

FONTAINE, A.B., and PETERSON, W.W., Group Code Equivalence and Optimum Codes, IRE Trans., IT-5, Special Supplement, 60-70 (1959).

GARNER, H.L., [1] Generalized Parity Checking, IRE Trans. on Electronic Computers, Vol. EC-7, 207-212, (1958).

GARNER, H.L., [2] A Ring Model for the Study of Multiplication for Complement Codes, IRE. Trans. on Electronic Computers, EC-8, p 25-30, (1959).

GILBERT, E.N., Gray codes and paths on the n-cube, Bell System Technical Publication, Monograph 3059, (1958).

GILBERT, E.N., and MOORE, E.F., Variable Length Binary Encodings, B.S.T.J. vol. 38, 933-968, (1959).

GREEN, J.H., and SAN SOUCIE, R.L., An Error-Correcting Encoder and Decoder of High Efficiency, Proc. IRE, vol 46, 1741-1744, 1958.

HAGELBARGER, D.W., Recurrent Codes: Easily Mechanized, Burst-Correcting and Binary Codes, B.S.T.J., vol. 38, 969-984, 1959.

HAMMING, R.W., Error-Correcting and Error-Detecting Codes, Bell System Tech. J., 29, 147, 1950.

HARTLEY, R.V.L., Transmission of Information, B.S.T.J., 7, p 535, 1928.

HUFFMAN, D.A., [1], The Synthesis of Linear Sequential Coding Networks, Proc. Symposium on Information Theory, London, 1955.

HUFFMAN, D.A., [2], A Method for the Construction of Minimum Redundancy Codes, Proc. IRE, 1098-1101, Sept. 1958.

I.B.M. Corp. [1], "IBM 7340 Hypertape Drive Reference Manual," Form A 22-6616, (1962).

I.B.M. Corp. [2], "Manual of Instruction. 1620 Data Processing System," Form 227-5507-1 (1960).

I.B.M. Corp. [3], "Reference Manual. IBM 1620 Data Processing System," Form A26-4500-2 (1959, 1960, 1961).

IVERSON, K.E., [1] "A Programming Language," John Wiley, New York, 1962.

IVERSON, K.E., [2], The Description of Finite Sequential Processes, in "Information Theory, Fourth London Symposium," Colin Cherry (Ed.), Butterworths, London, 1961.

IVERSON, K.E., [3] Formalism in Programming Languages, Research Report (unpublished) Harvard Univ., July, 1963.

KAUTZ, W.H., A class of multiple-error-correcting codes, University of Michigan Engineering Summer Conferences, 1962.

KHINCHIN, A.I., "Mathematical Foundations of Information Theory," Dover Publications, New York, 1957 (Tr. by R.A. Silverman and M.D. Friedman).

LAEMMEL, A.E., Efficiency of Noise-Reducing Codes, in "Communication Theory," W. Jackson (Ed.) 111-118, Academic Press, Inc., New York, 1953.

LLOYD, S.P., Binary Block Coding, B.S.T.J. vol. 36, p 517, 1957.

LEBOW, I.L., Communication in Digital Systems, in "Information Theory, Fourth London Symposium," Colin Cherry, (Ed.), Butterworths, London, 1961.

LEDLEY, R.S., "Digital Computer and Control Engineering," McGraw-Hill, New York, 1960.

McLUSKEY, E.J., Error-Correcting Codes - A Linear Programming Approach, B.S.T.J. Vol. 38, 1485-1512, (1959).

McMILLAN, B., Two Inequalities Implied by Unique Decpherability, IRE Trans. on Inf. Theory, Vol. IT-2 115-116, Dec. 1956.

MEGGITT, J.E., [1] Error Correcting Codes for Correcting Bursts of Errors, IBM J. of Research & Dev., Vol. 4, p 329-334, July, 1960.

MEGGITT, J.E., [2] Error-Correcting Codes and Their Implementation for Data Transmission Systems, IRE Trans. on Inf. Theory, Vol. IT-7, No. 4, Oct. 1961.

MINNEAPOLIS-HONEYWELL Co., "Honeywell 400, General Information Manual," p 83-87, (1962).

ORE, O., "Graphs and their Uses," Random House, Inc., New York, 1963.

PETERSON, W.W., "Error Correcting Codes," M.I.T. Press and John Wiley and Sons, Inc., New York, 1961.

PHISTER, M., "Logical Design of Digital Computers," John Wiley & Sons, 1958.

PRANGE, E., Some Cyclic Error-Correcting Codes with Simple Decoding Algorithms, Air Force Cambridge Research Center, TN -48 -156, ASTIA DOC. No. AD 1523P6 (1958).

REZA, F.M., "An Introduction to Information Theory," McGraw-Hill, New York, 1961.

REED, I.S., A Class of Multiple-error Correcting Coding and Decoding Schemes, IRE Trans. on Inf. Theory, Vol. IT-4 38-49, 1954.

SARDINAS, A.A., and PATTERSON, G.W., A Necessary and Sufficient Condition for Unique Decomposition of Coded Messages, IRE Conv. Record, pt. 8, 104-109, March 1953.

SCOTT, N.R., "Analog and Digital Computer Technology," McGraw-Hill, 1960.

SHANNON, C.E. [1], A Mathematical Theory of Communication, B.S.T.J. Vol. 27, 379-423, 623-656, 1948.

SHANNON, C.E. [2], Communication in the Presence of Noise, Proc. ITE, Vol. 37, 10-21, 1949.

SILVERMAN, R.A., and CHANG, S.H., IRE Trans. on Inf. Theory, Vol. IT-4, 153, 1958.

SLEPIAN, D., [1] A Class of Binary Signalling Alphabets, B.S.T.J., Vol. 35, 203-234 (1956).

SLEPIAN, D., [2], A Note on Two Binary Signalling Alphabets, IRE Trans., IT-2 84-86 (1956).

SLEPIAN, D., [3] Some Further Theory of Group Codes, B.S.T.J., Vol. 39, 1219-1252 (1960).

TOMPKINS, H.E., Unit-distance Binary Codes, University of Michigan Engineering Summer Conference, 1962.

von NEUMANN, J., Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components, "Automata Studies," Ann. of Math. Studies, No. 34, Princeton Univ. Press, Princeton, N.J., 1956.

WEEG, G.P.: Uniqueness of Weighted Code Representation, I.R.E. Trans. on Electronic Computers, Vol. EC-9, No. 4, pp. 487-490, Dec. 1960.

WOLFOWITZ, L., "Coding Theorems of Information Theory," Printice-Hall, Englewood Cliffs, N.J., 1961.

WOZENCROFT, J.M., and REIFFEN, B., "Sequential Decoding," Technology Press of M.I.T., and John Wiley, New York, (1961).

ZIERLER, N., Several Binary-Sequence Generators, M.I.T. Lincoln Lab. Technical Report No. 95, 1955.